Part 1 – Example 1

First Steps of any analysis

- Snapshot your vm before beginning any analysis
- Hash all files that you plan to analyze
- Keep a log of steps taken

Opening Comments

- Easy to determine maliciousness of malware
- But to determine the purpose we must be willing to analyze on a more granular level
- This allows better IOC's and stronger understanding of the purpose and behavior

Analysis Routine

1)Static Analysis
- Strings
- Imports
- Exports

2)Dynamic Analysis
- Set breakpoints
- Analyze data, memory, and functions at runtime

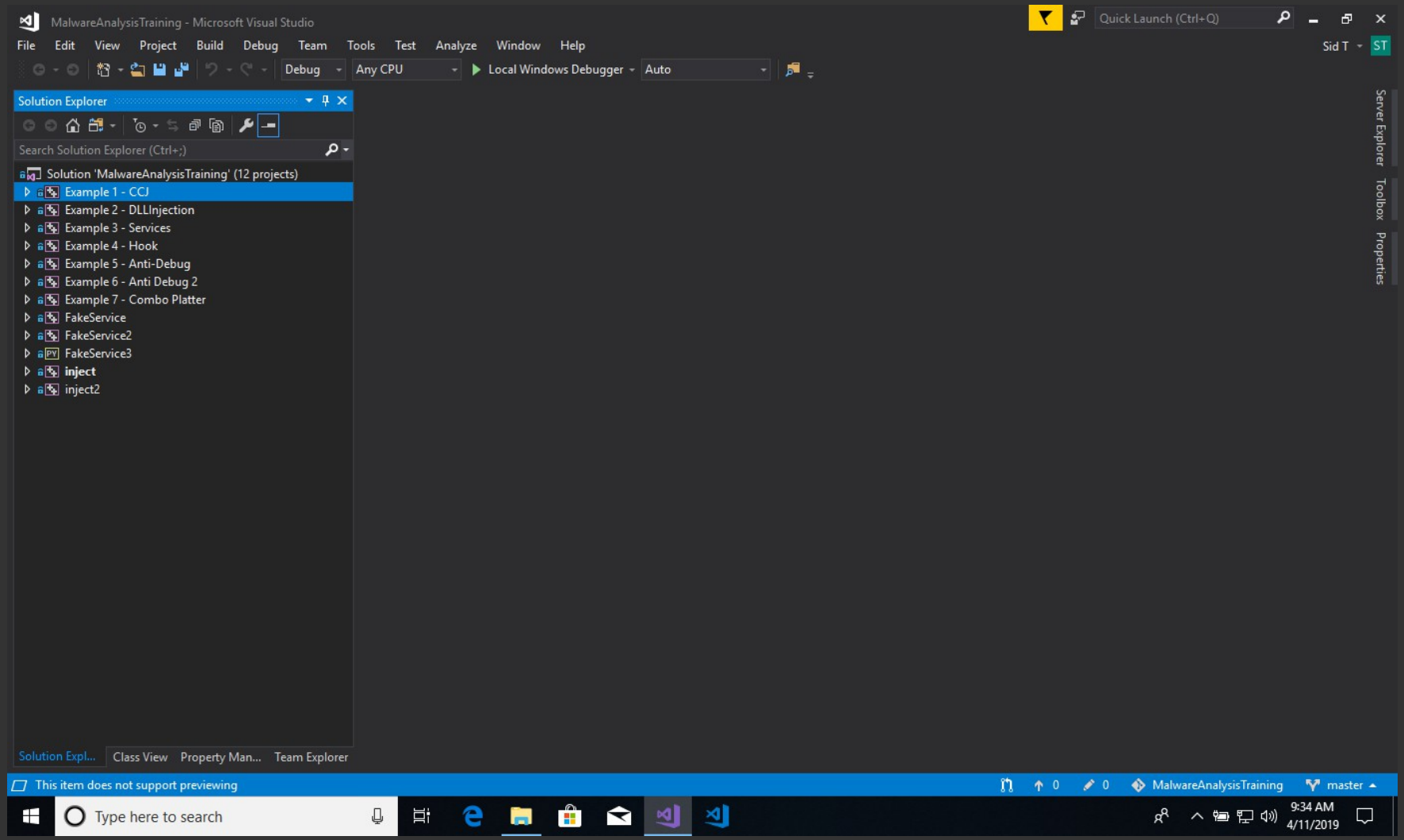
3)Update your timeline.

4)Back to Step 1.

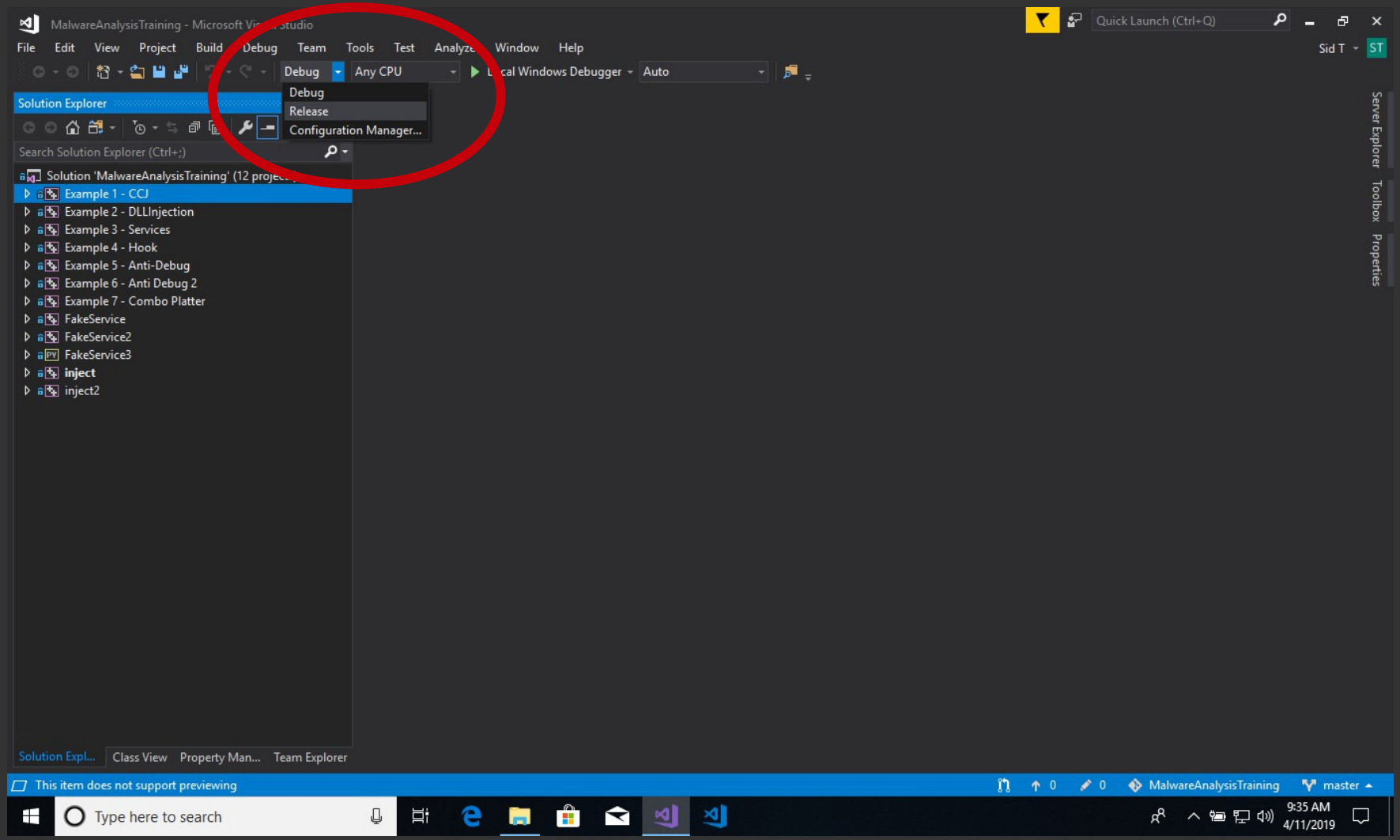
Setting up the First Example

(The Windows VM will be required at this point with the project opened and ready in Visual Studio. Read the Requirements.pdf for setting up the project.)

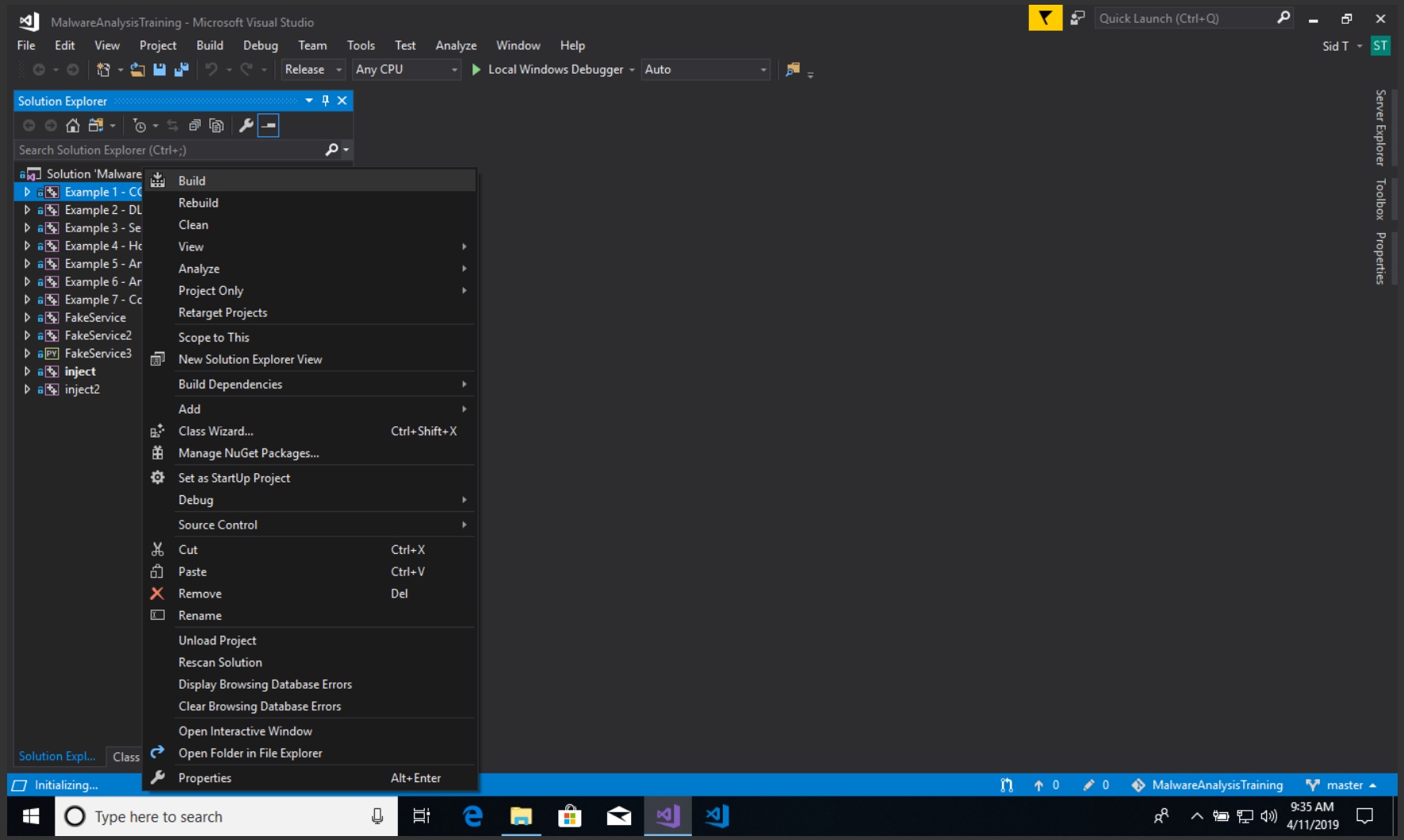
This is what the Visual Studio project should look like if opened up properly.



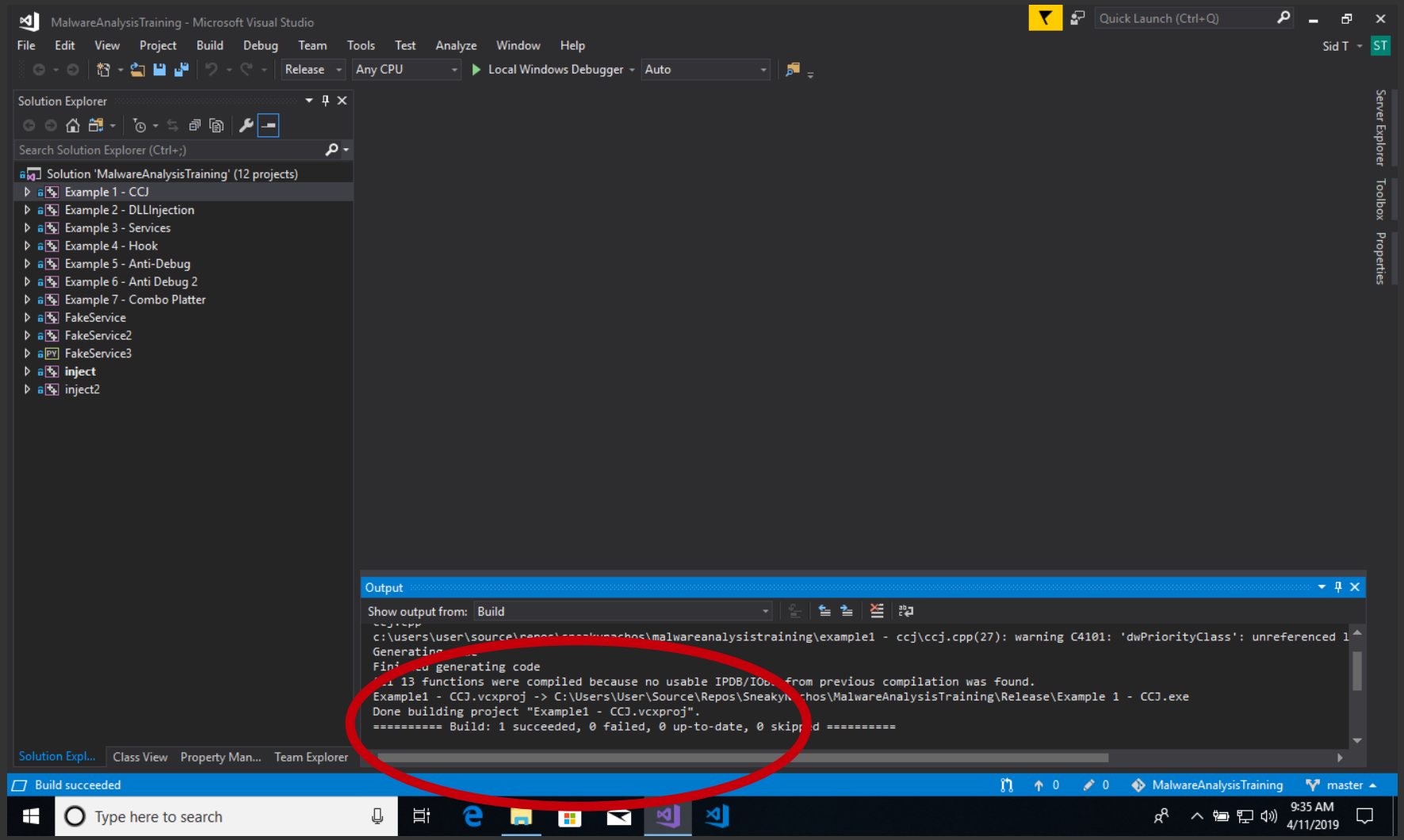
We first need to set the build to Release. Shown below is where the drop down is to switch between Debug/Release Builds. “Any CPU” should also be set to the target architecture of the system (x86) if 32-bit and (x64) if 64-bit.



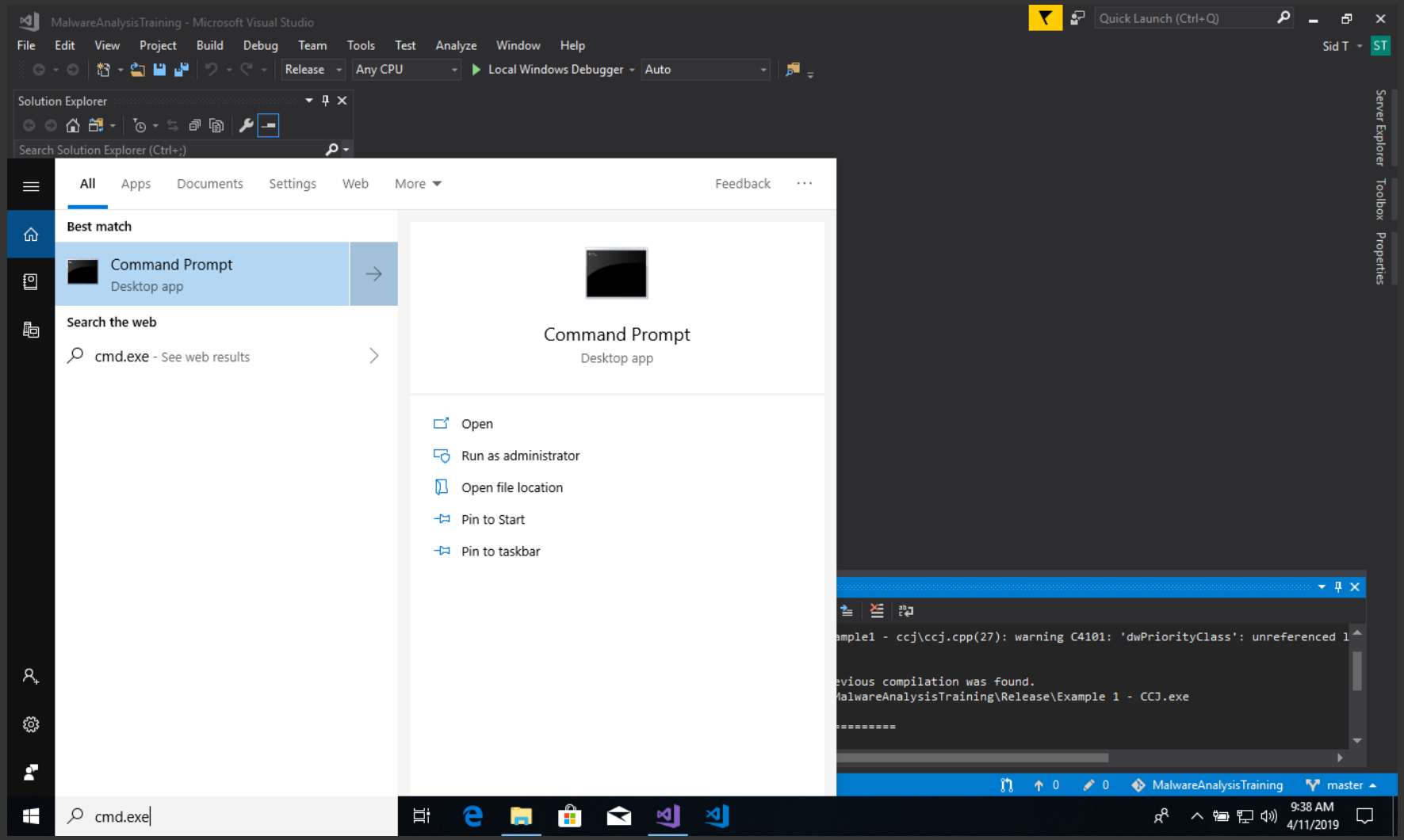
Hover over “Example 1 – CCJ” and right click to open up the various drop down options for the project. At the top there will be a “Build” option. Go ahead and click build and wait for the compiler to startup.



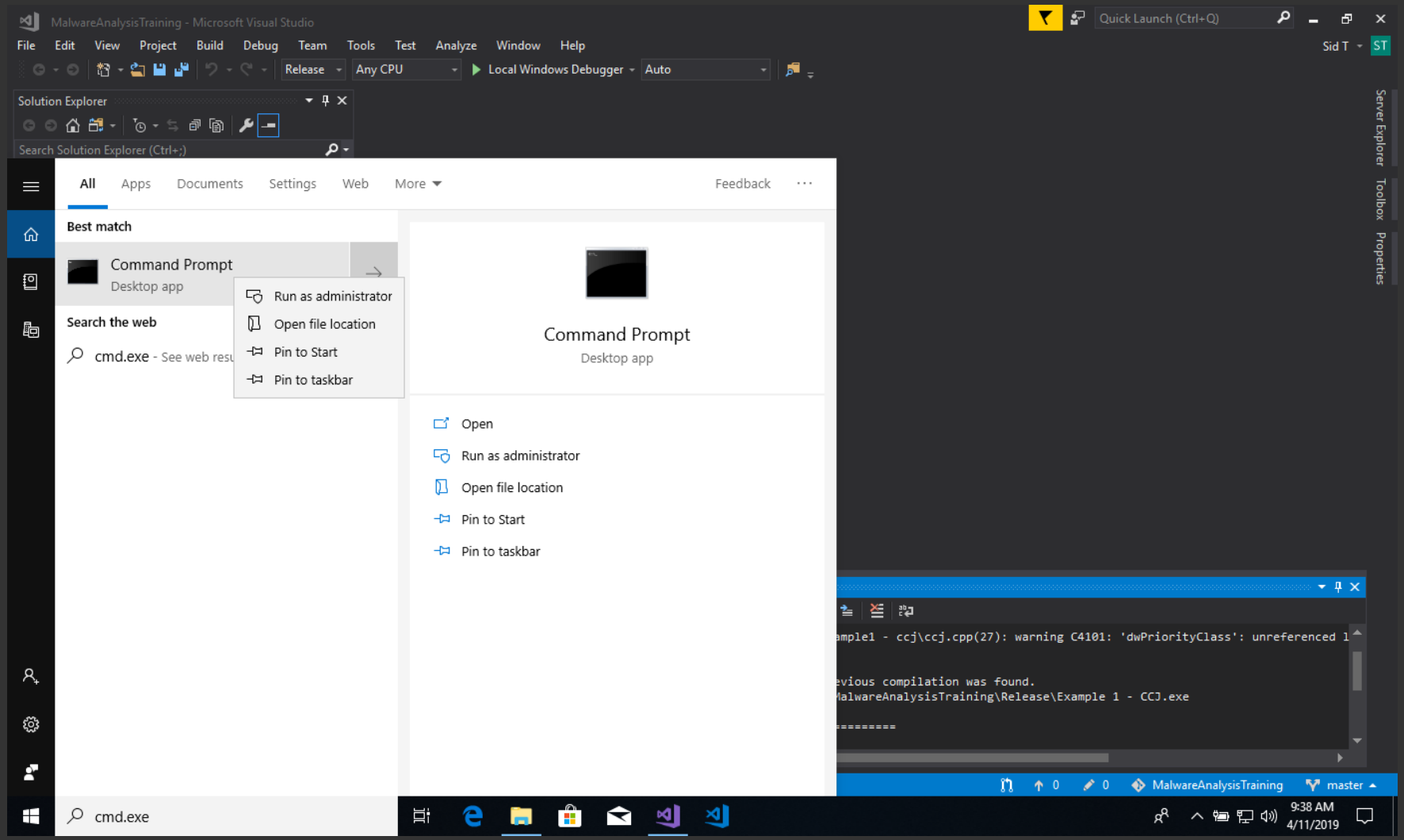
You should see success like the example below in the output window circled in red.



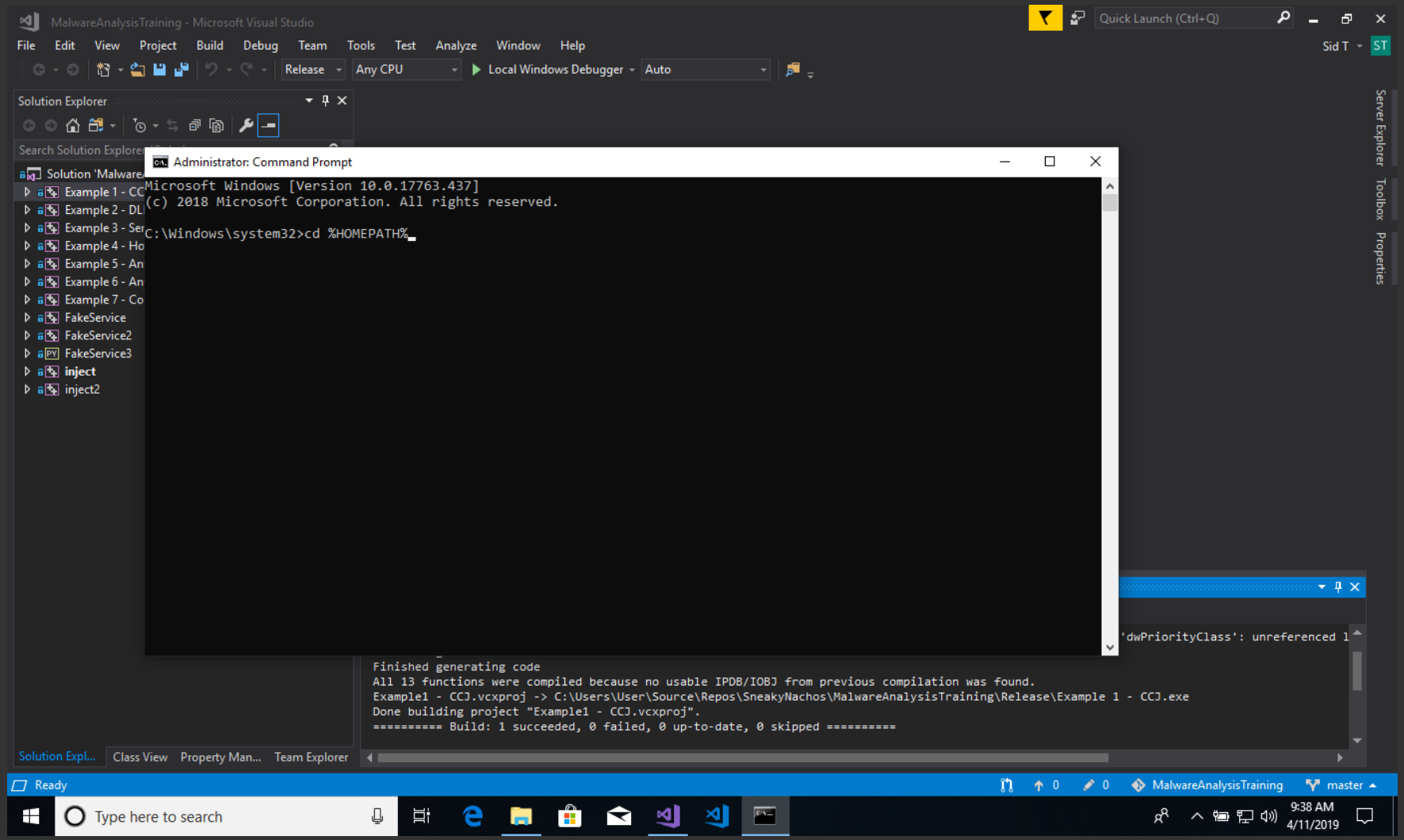
Let's go ahead and open up a command prompt running as Administrator.



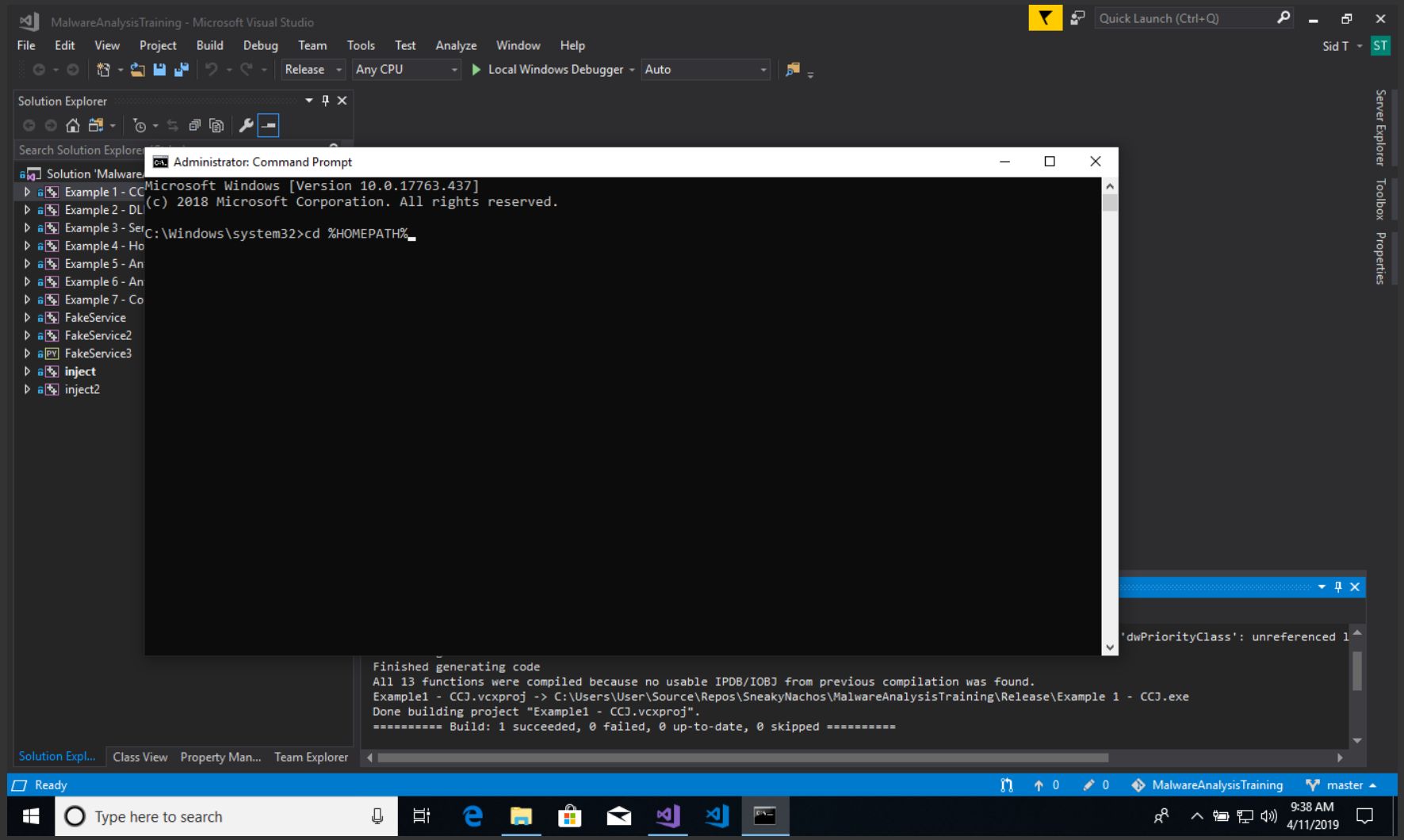
Remember to right click the command prompt to get the “Run as administrator” option.



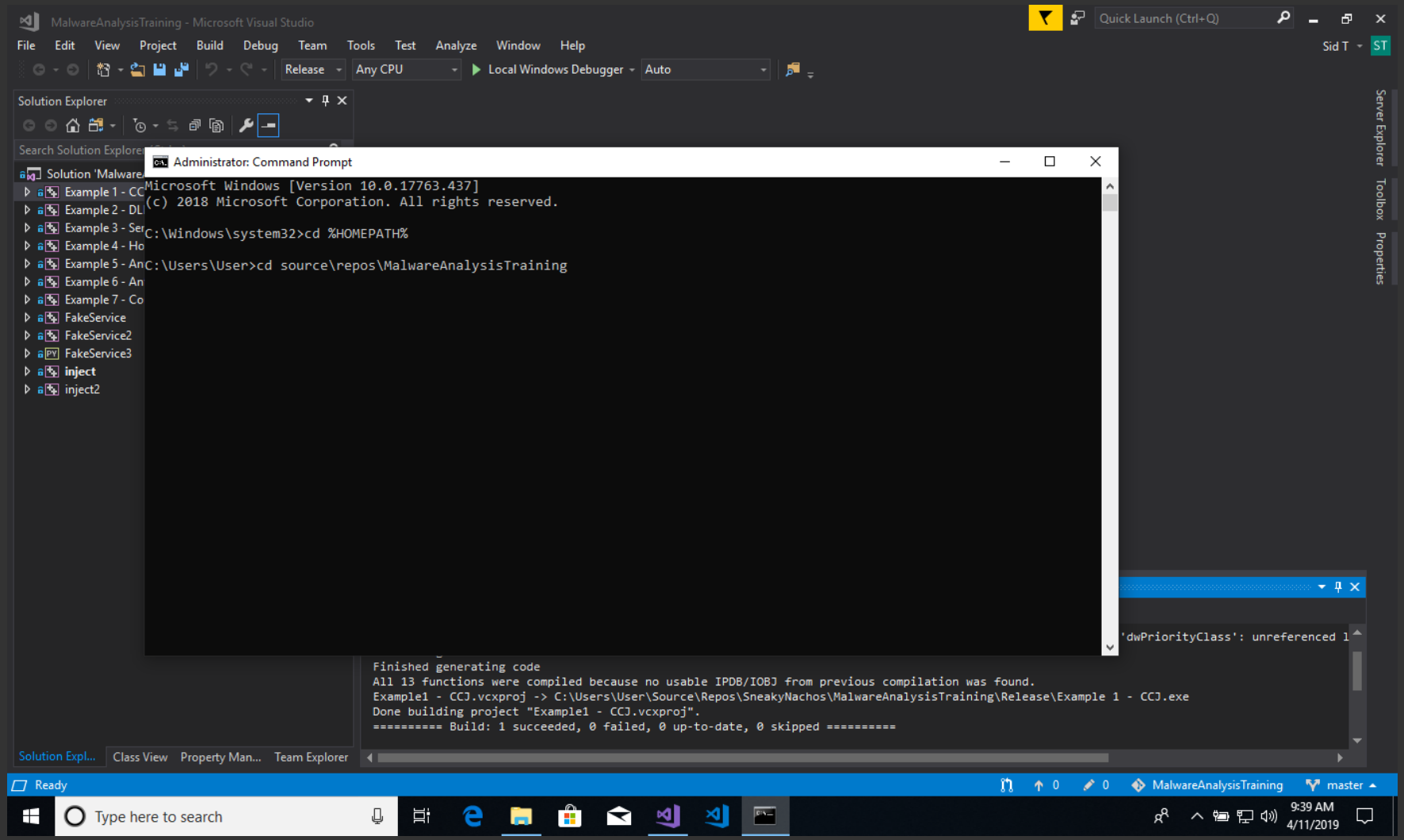
Now let us change to the directory of the project that you downloaded.



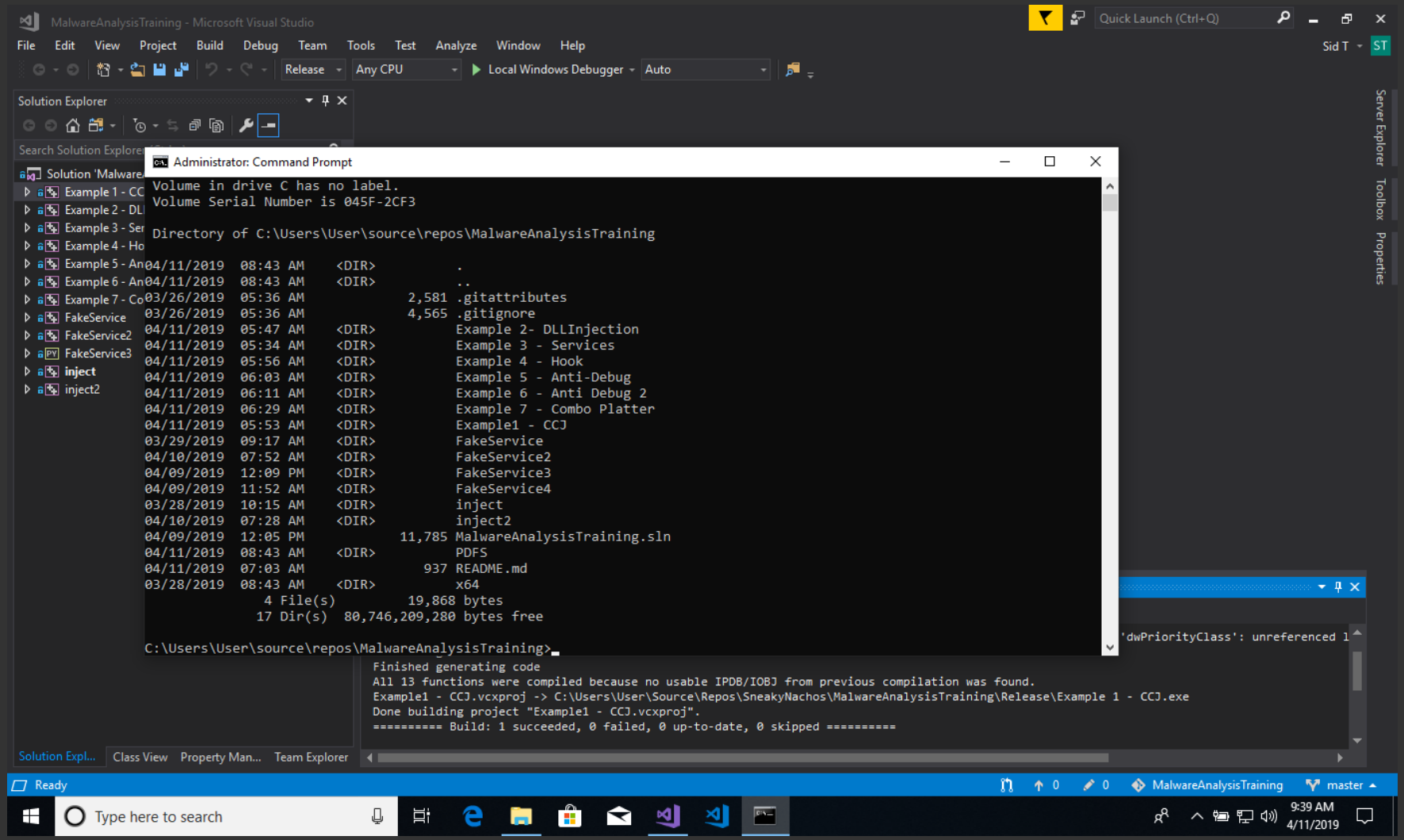
Now let us change to the directory of the project that you downloaded.



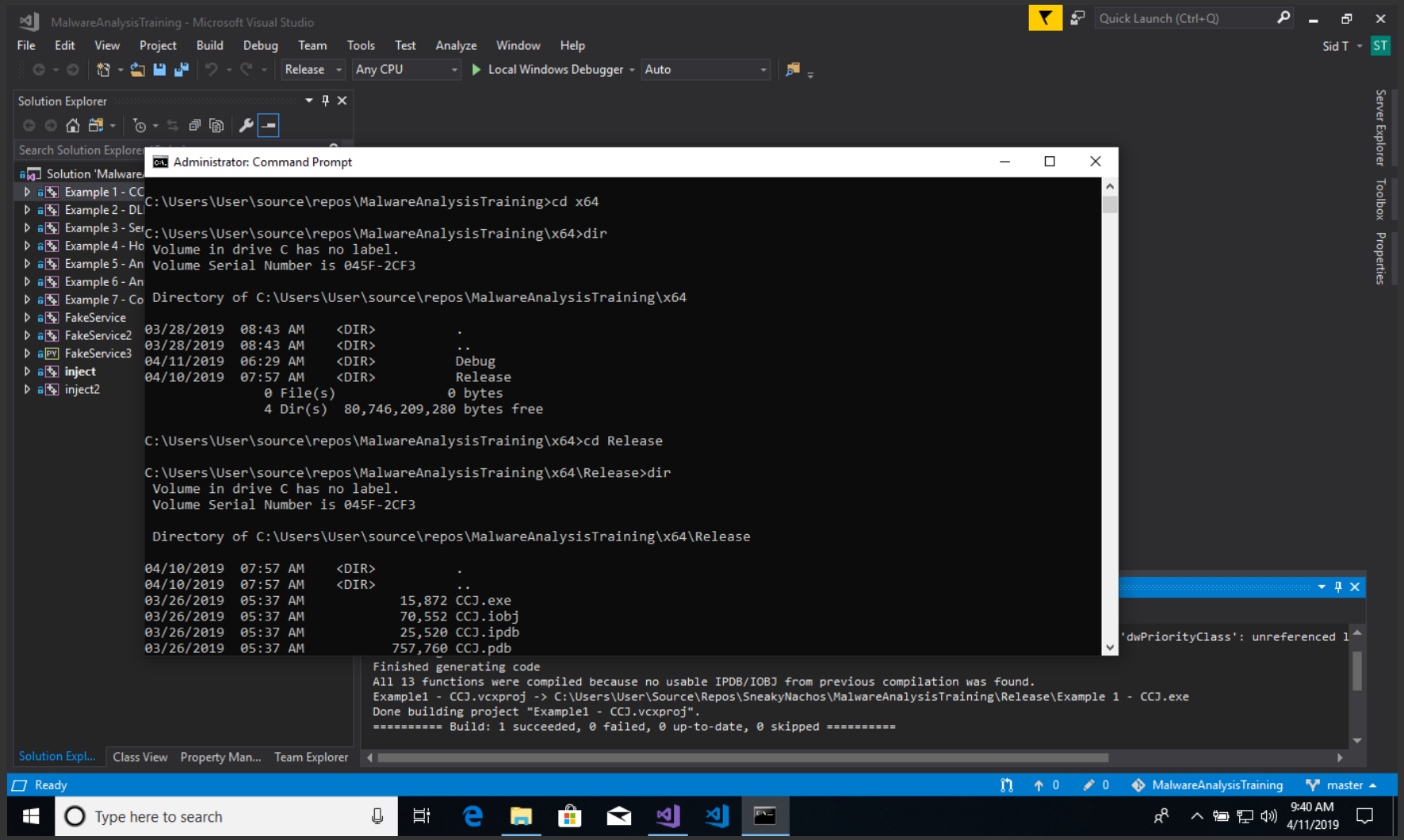
Note yours is probably on the Desktop, whereas the one built in the slides is from the source repo.



Now sitting in the highest level of the directory, after you've hit the “Build” button a new folder should show up. Either it will be the “x64” folder or the “Release” folder. If “x64”, change into the “x64” directory and change into the “Release” folder within it. If you already have a “Release” folder at the top, go ahead and change into that directory.



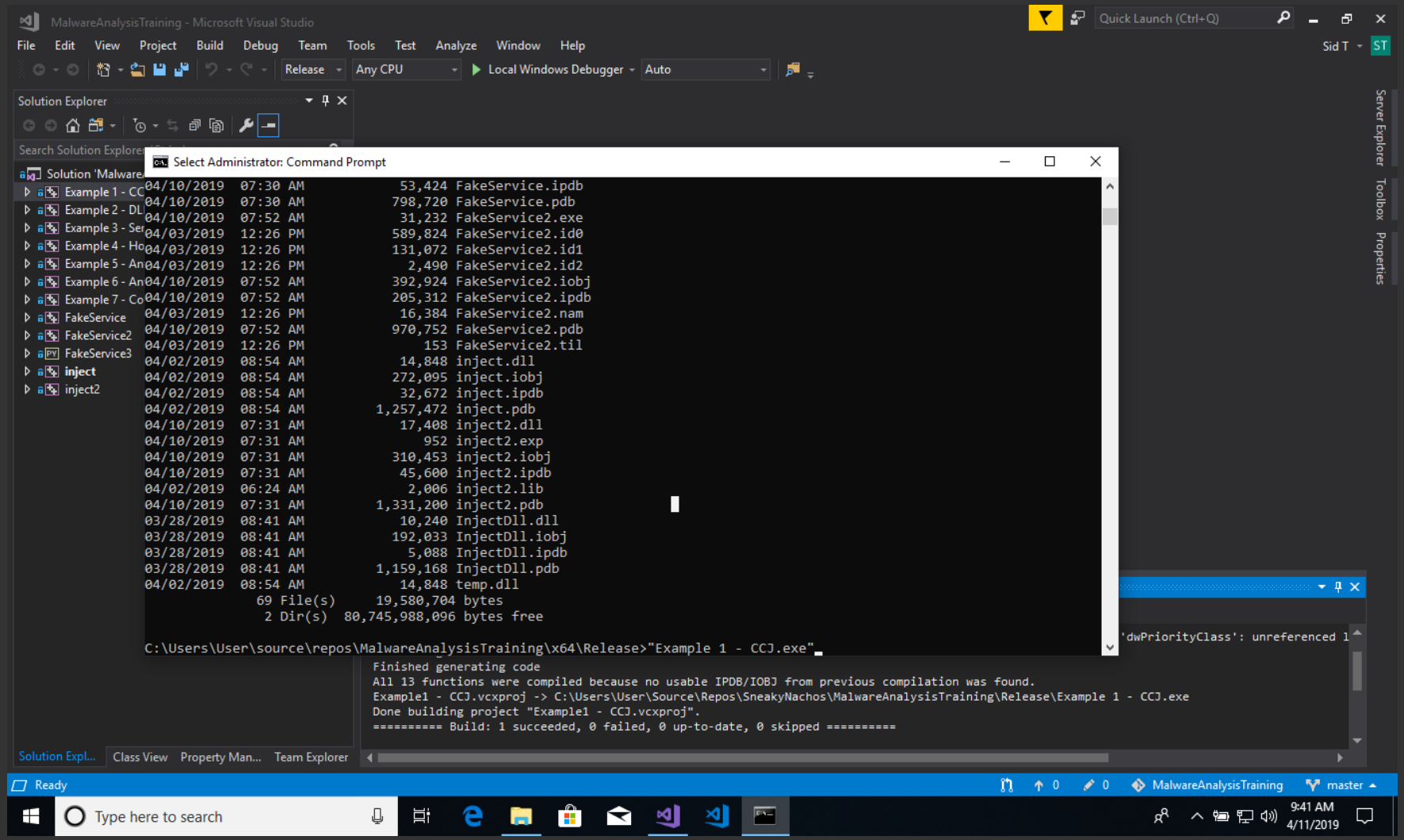
When looking inside the “Release” folder, by typing “dir” you should see your example 1 binary built. Depending on when you downloaded the examples it could be “CCJ.exe” or “Example 1 – CCJ.exe”.



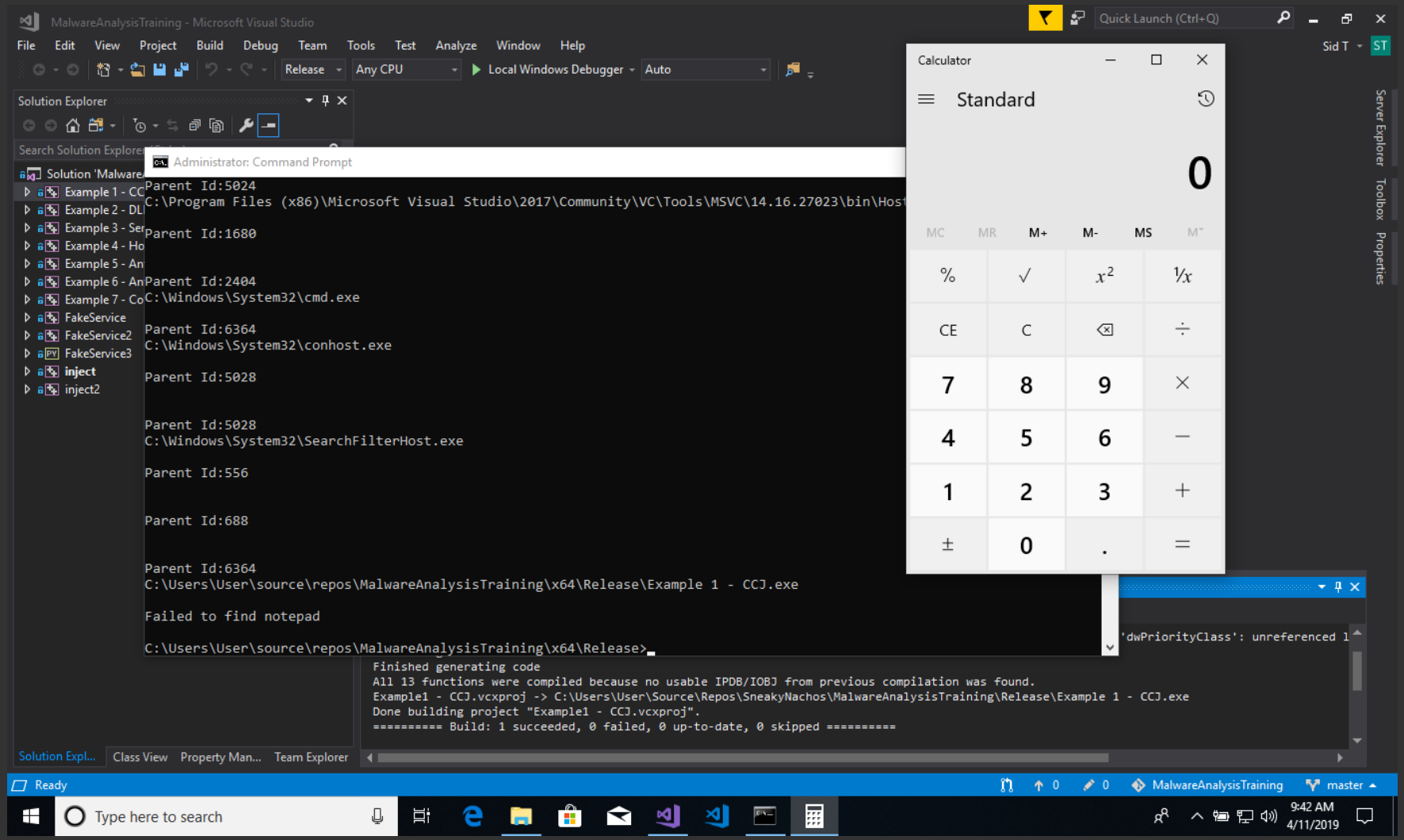
Common Issues

- The “Requirements.pdf” should assist you if your really stuck and out of options with compiling.
- Remember to delete the AppData/Local and AppData/Roaming for Visual Studio
- Make sure the “Any CPU” option is set to the correct target. “Any CPU” defaults to 32-bit which may be wrong.
- You may have not installed the C++ libraries for visual studio. (See Requirements.pdf)

Let's go ahead and do something dumb first. We'll just run the example first and see what the result is and define our initial timeline from that.



Looks like a number of things happened. First a bunch of process names were spat out. “Failed to find notepad” was printed, and a calculator spawned from nowhere. So at least now we have a general basis for the timeline.



Goals of Example 1

- You should be able to give me a general analysis and behavior timeframe of what the binary is doing.
- This includes the start all the way to end.

Staring the Initial Analysis

- The first step is a brief static analysis.
- Reasons for starting the static analysis first.
  1) Easily acquired meta data may give us insight into the behavior/purpose .
  2) Makes dynamic analysis faster.
  3) Reduce surface area of the analysis.

First Step - Strings

A “String” is a combination of bytes that is directly followed by a NULL byte, aka zero in memory.

As an example the word “CAT” in memory is actually encoded as “0x43 0x41 0x54 0x00 “.

Where 0x43 is the hex encoded form of “C”, followed by “0x41” A, and “0x54” T. Which is then finished off by the “0x00” aka the NULL byte. We'll talk about memory arrangement later (Little endian/Big Endian).

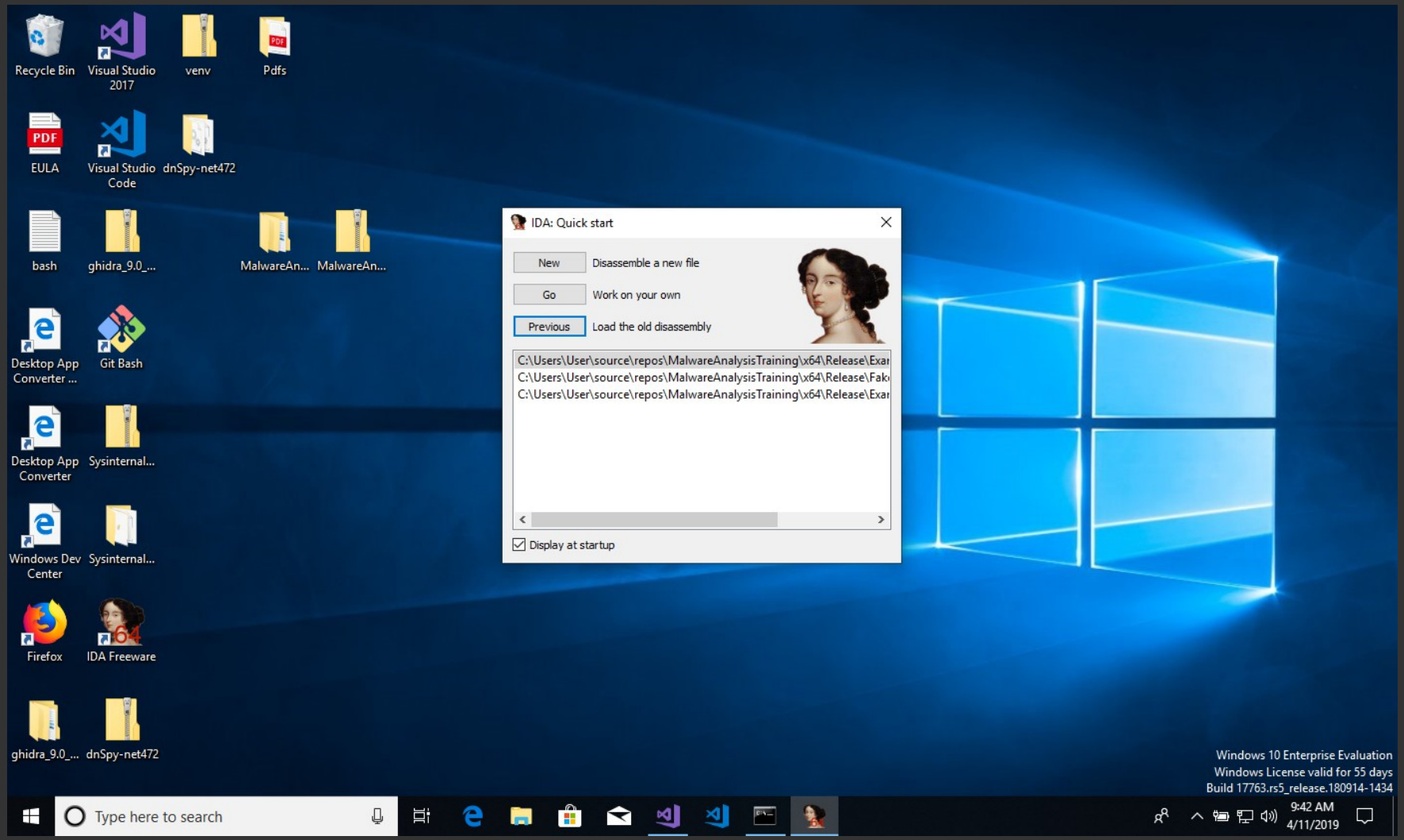
Why “Strings”?

Strings build a context of what the programmer wishes to do, is about to do, and possibly what didn't. Programmers use strings for printing, interacting with api's, debugging, web requests, error handling, file move/delete/create, data movement, and much more.

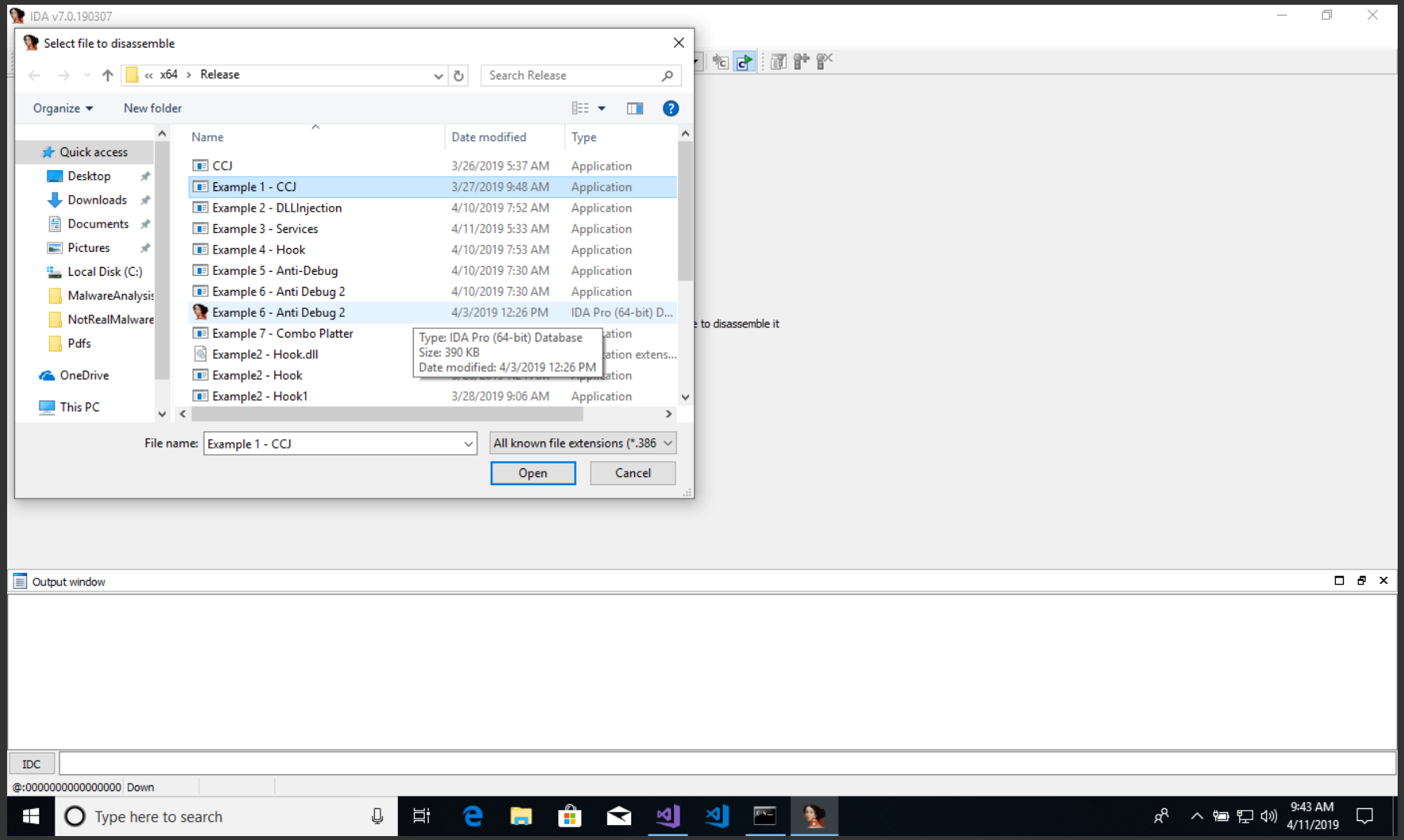
Why “Strings”?

This is very interesting to us as analysts because quite a number of the Windows API require strings to perform functionality. Thus if you can capture or find the string that is entering the Windows API and the Windows API function the malware wishes to make, you can generalize the action that the malware is trying to perform.

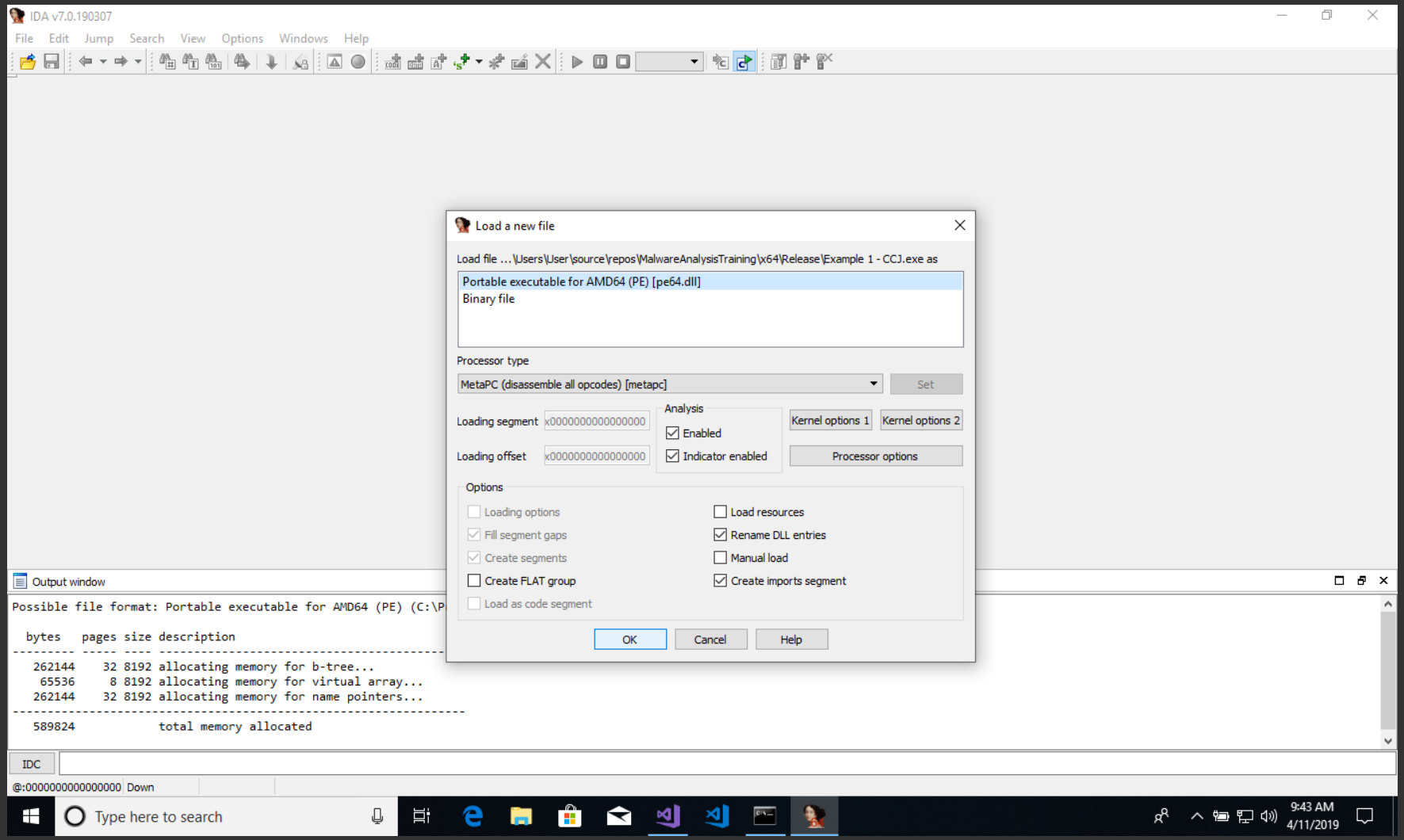
Now to look at the “Strings”, we'll pull up IDA Freeware and open the “.exe” in the Release folder.



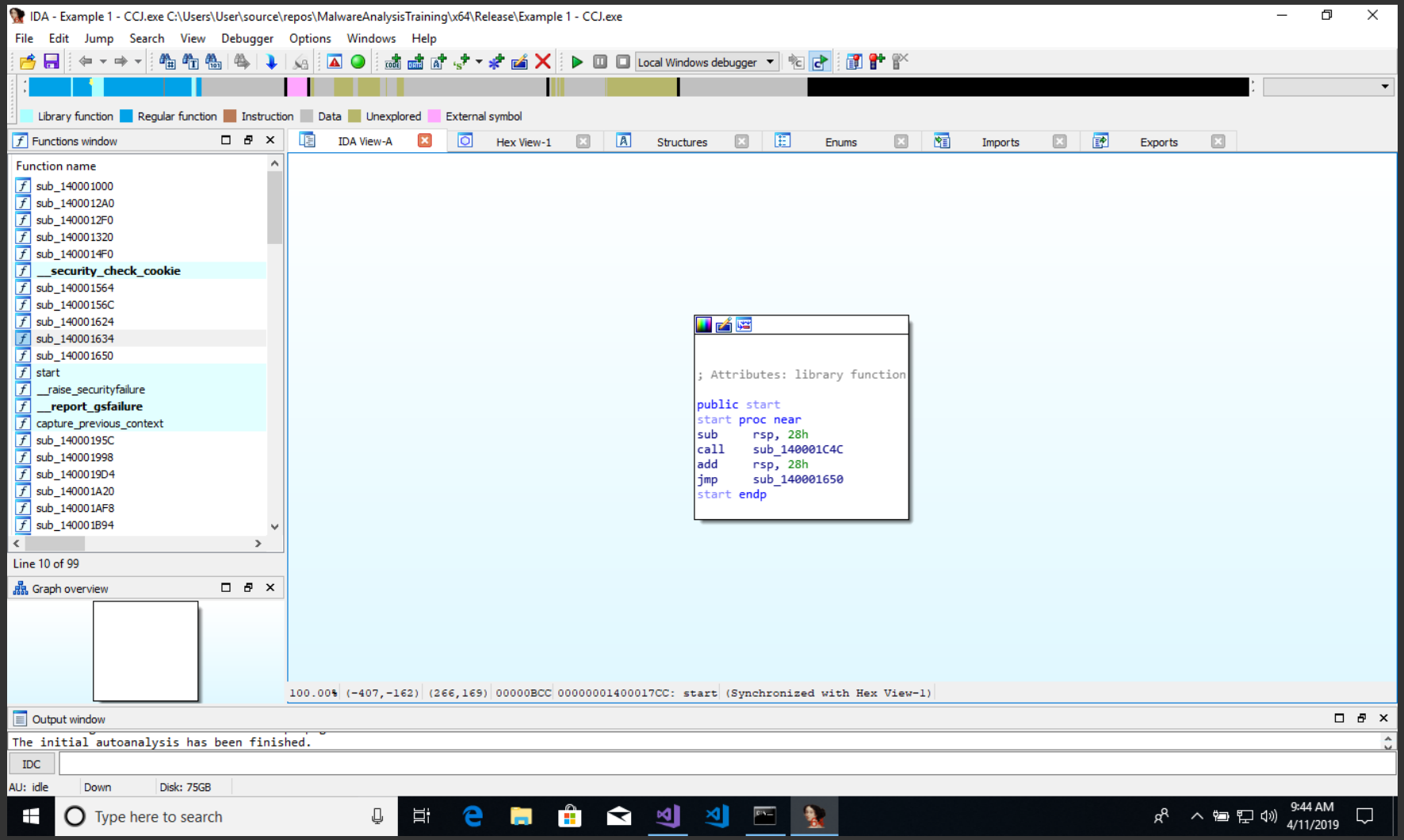
Open our “Example 1- CCJ” sitting in the Release folder.



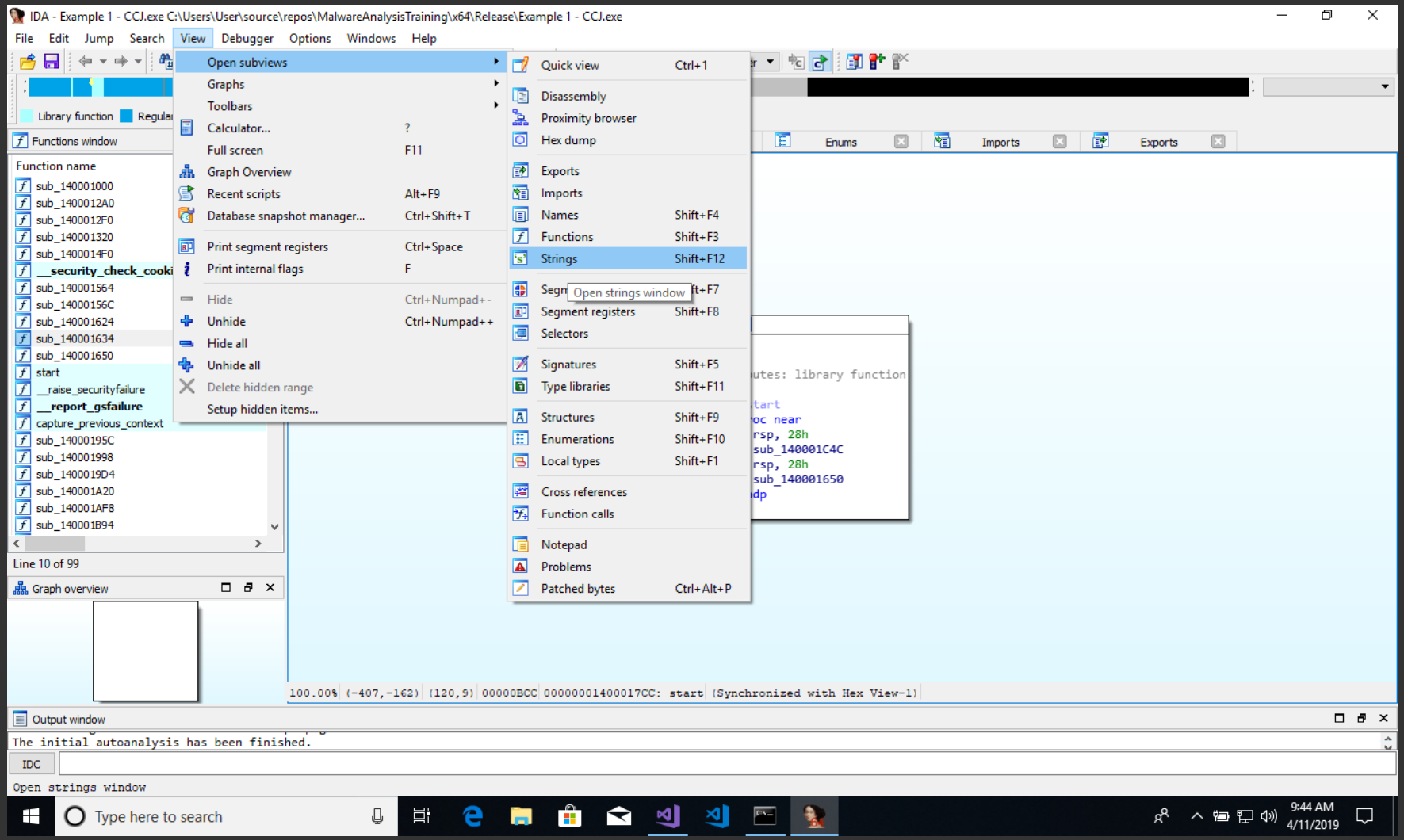
Don't worry about modifying anything when opening the file. All defaults are fine for this example.



This is what IDA should look like once you've successfully opened the example.

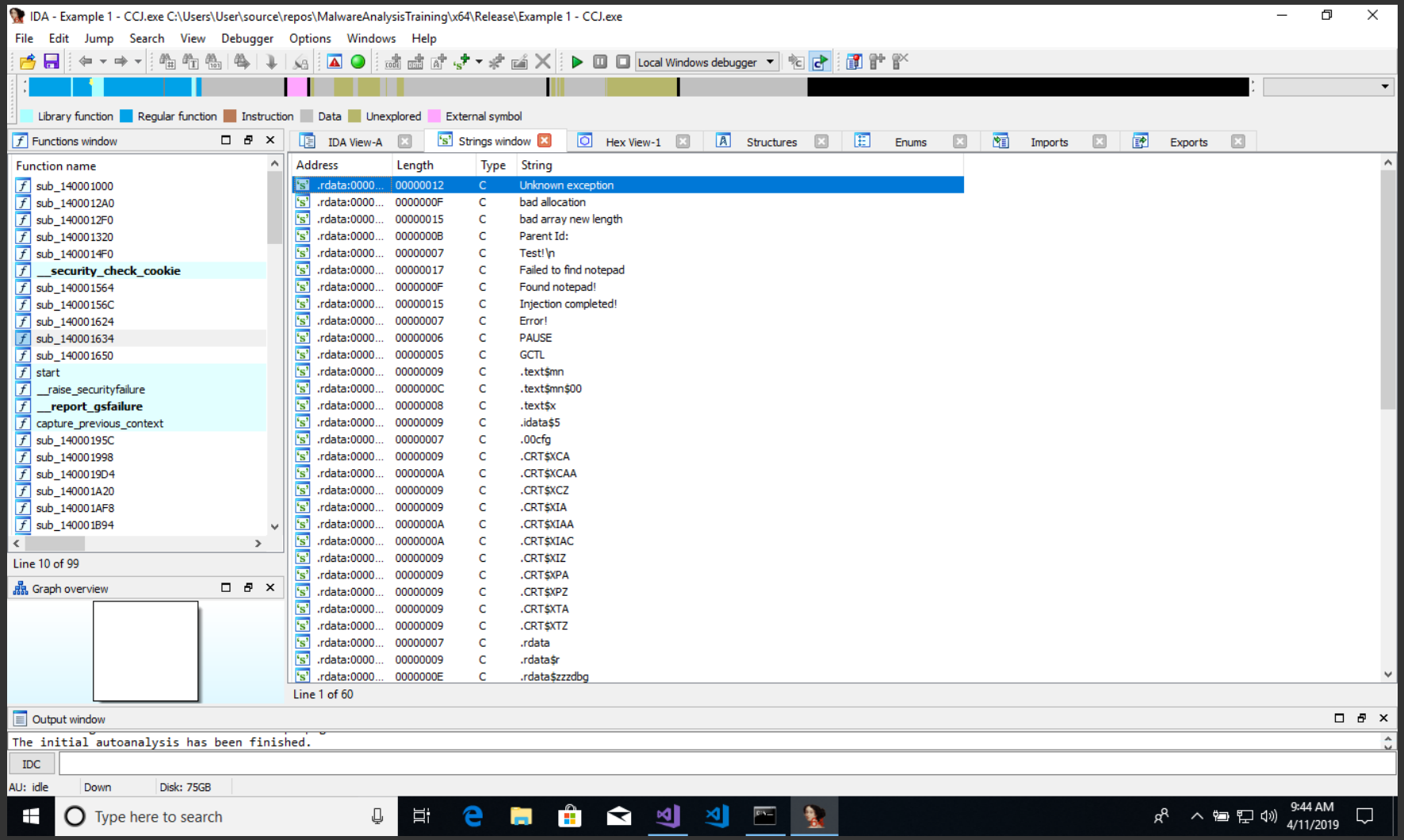


To open the strings sub-view, there's a tab at the top called “View”. Click the “View” drop down and hover over “Open subviews” and go down to “Strings” and click it.



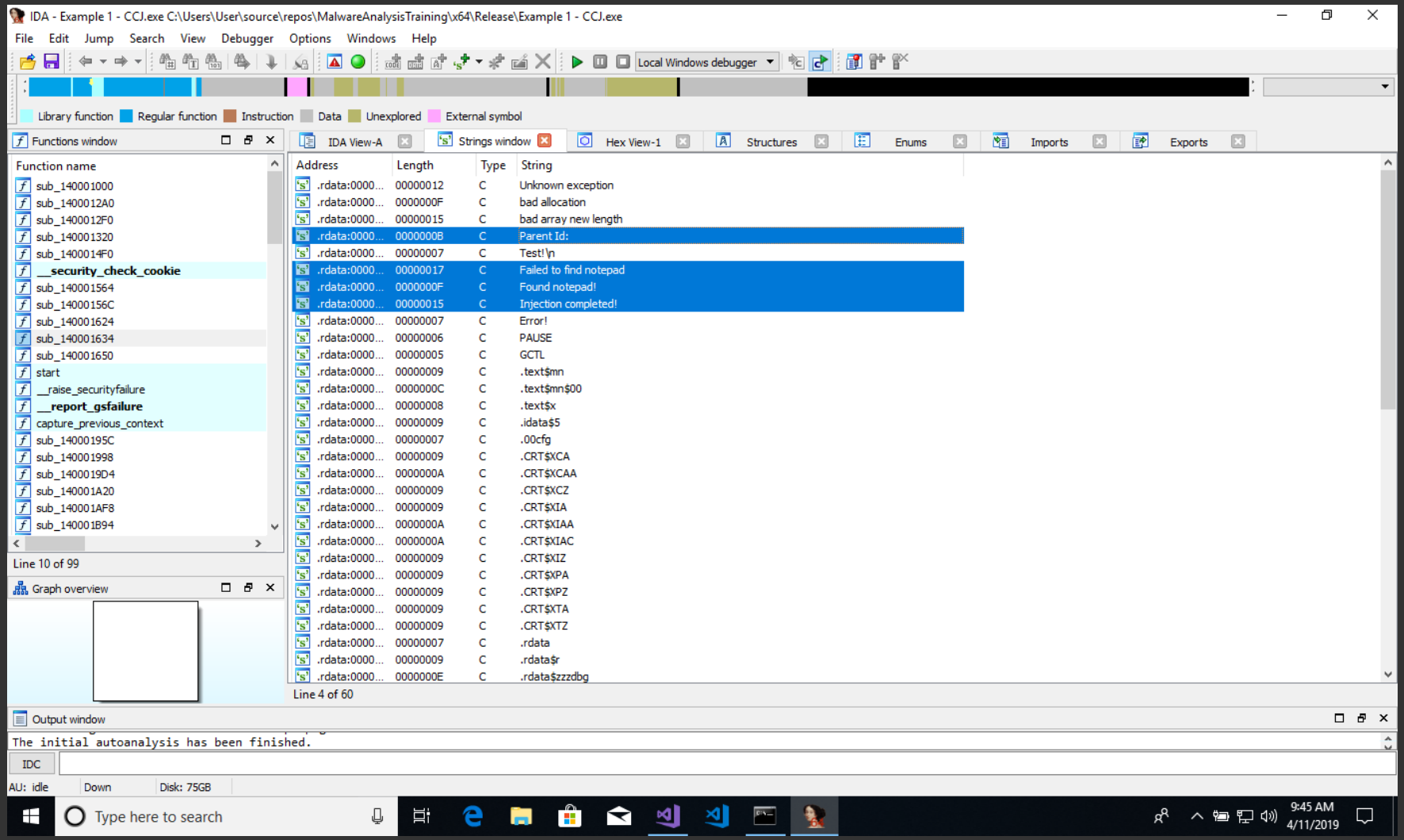
A brand new sub-view should pop up, and we should now see all the strings in the example.

First things first. We want to note all interesting strings that we with to analyze first.



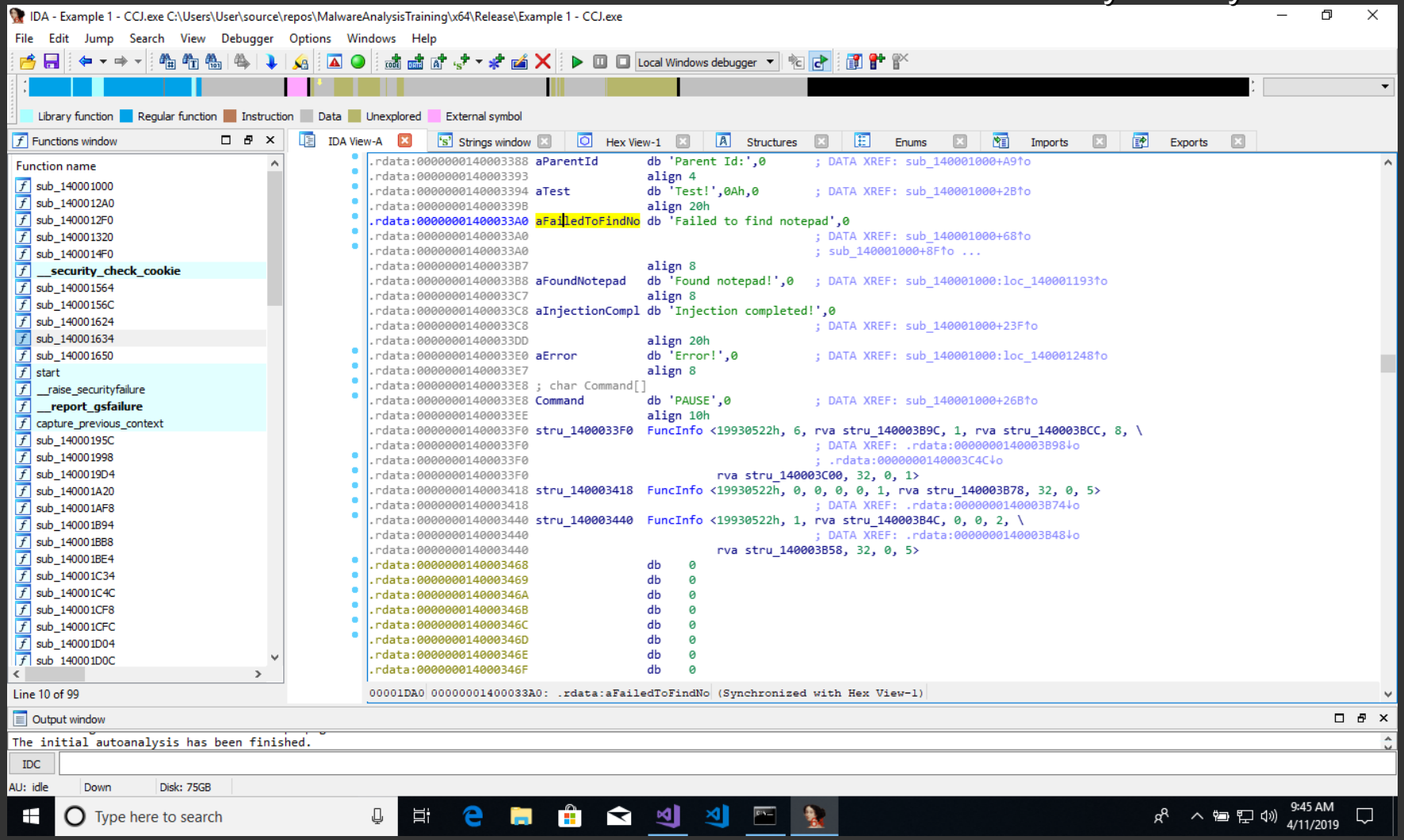
I've highlighted in blue the ones that I found particularly interesting to this example. “Failed to find notepad” was what I considered the highest on my list because it printed out during the running of the binary from earlier.

Let's double click “Failed to find notepad” now.



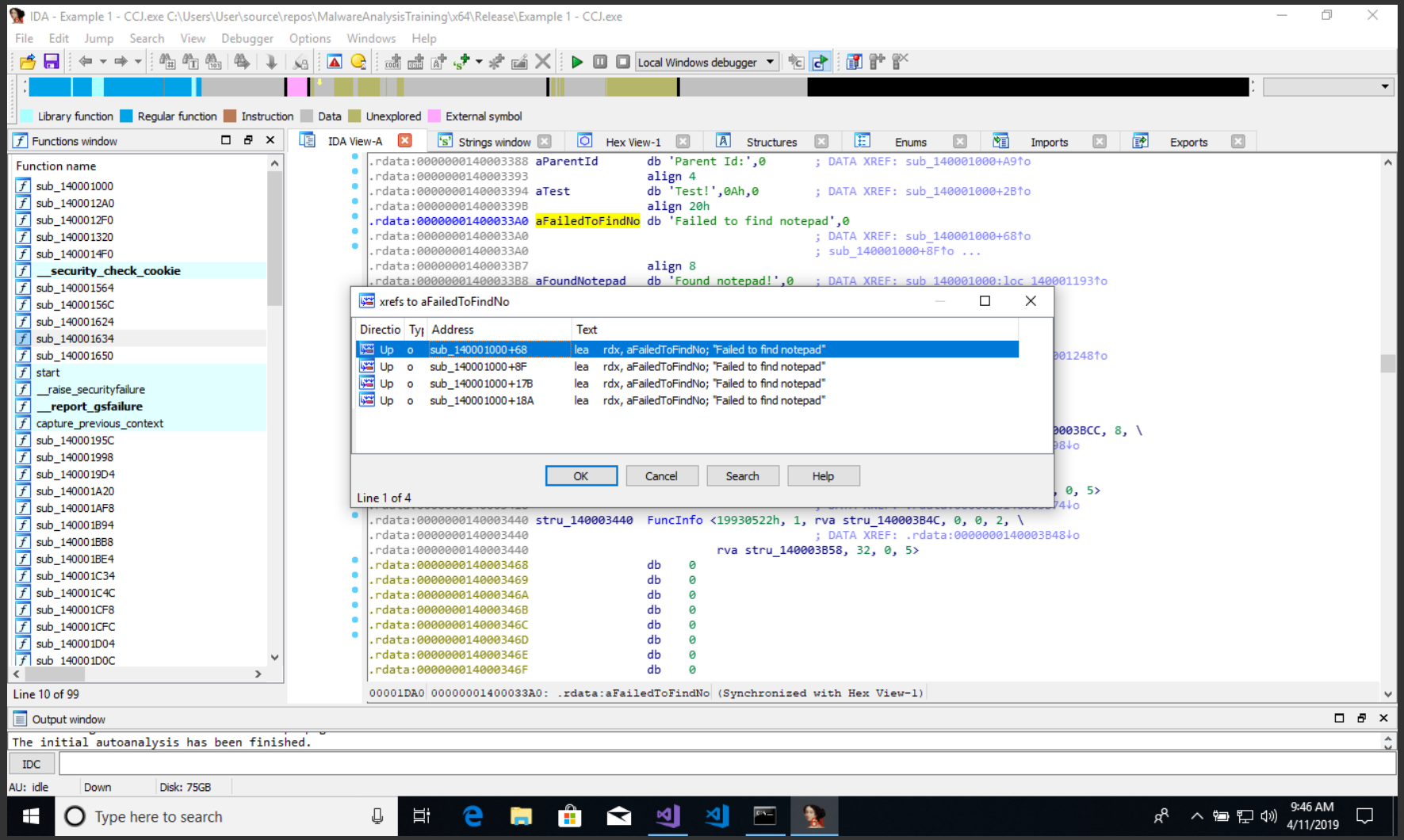
You'll be taken to this section if you double clicked the “Failed to find notepad” string. This is the .rdata section and is filled with read only data. IDA gave database names to the different strings. I highlighted to the right of the string that database name.

Now click the “aFailedToFindNo” name and hit the “X” button on your keyboard.



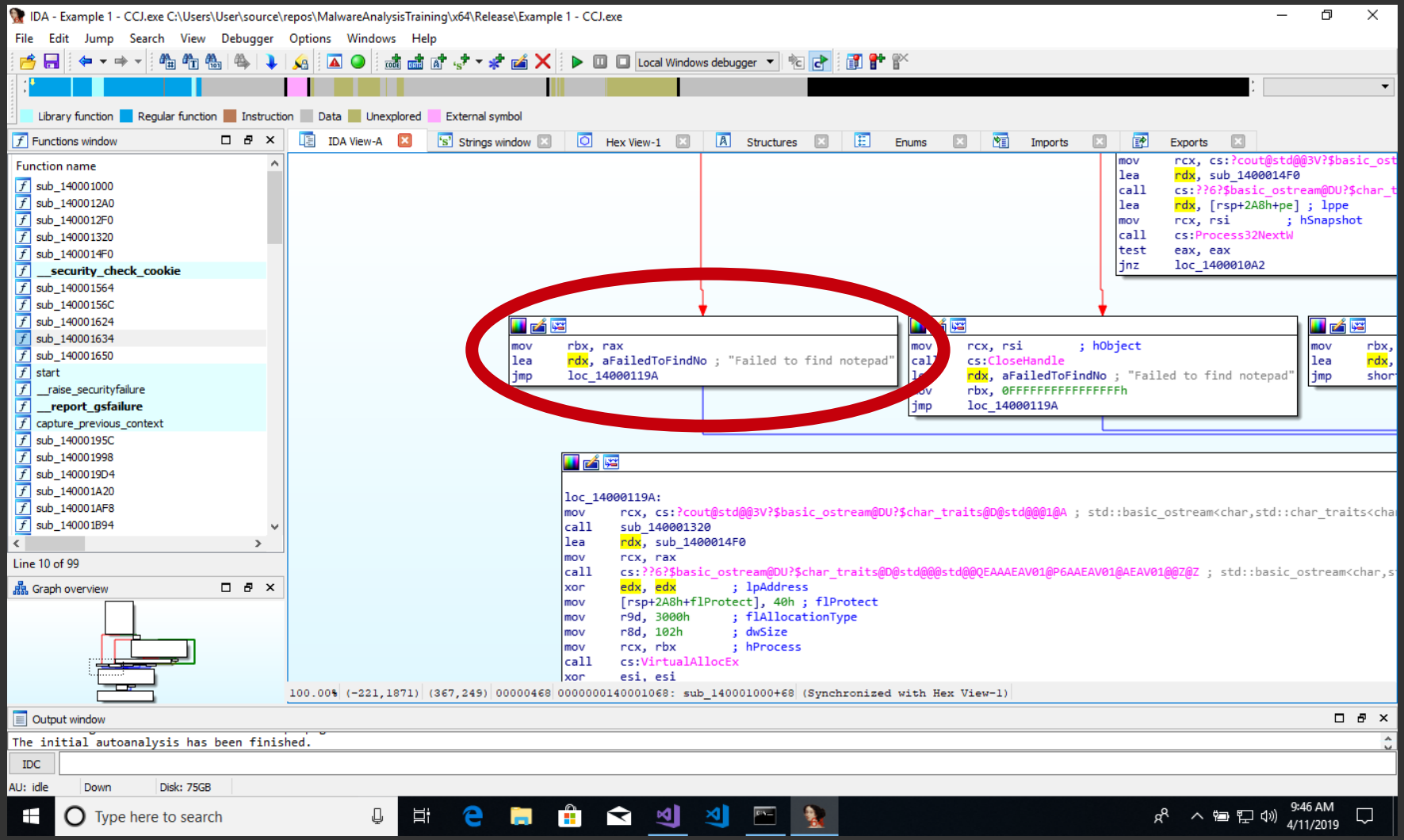
Hitting the “X” on the IDA database name should bring up the cross references windows. This allows us to track in the assembly where the string “Failed to find notepad” is being used.

Let's double click the first option that I highlighted in blue below.



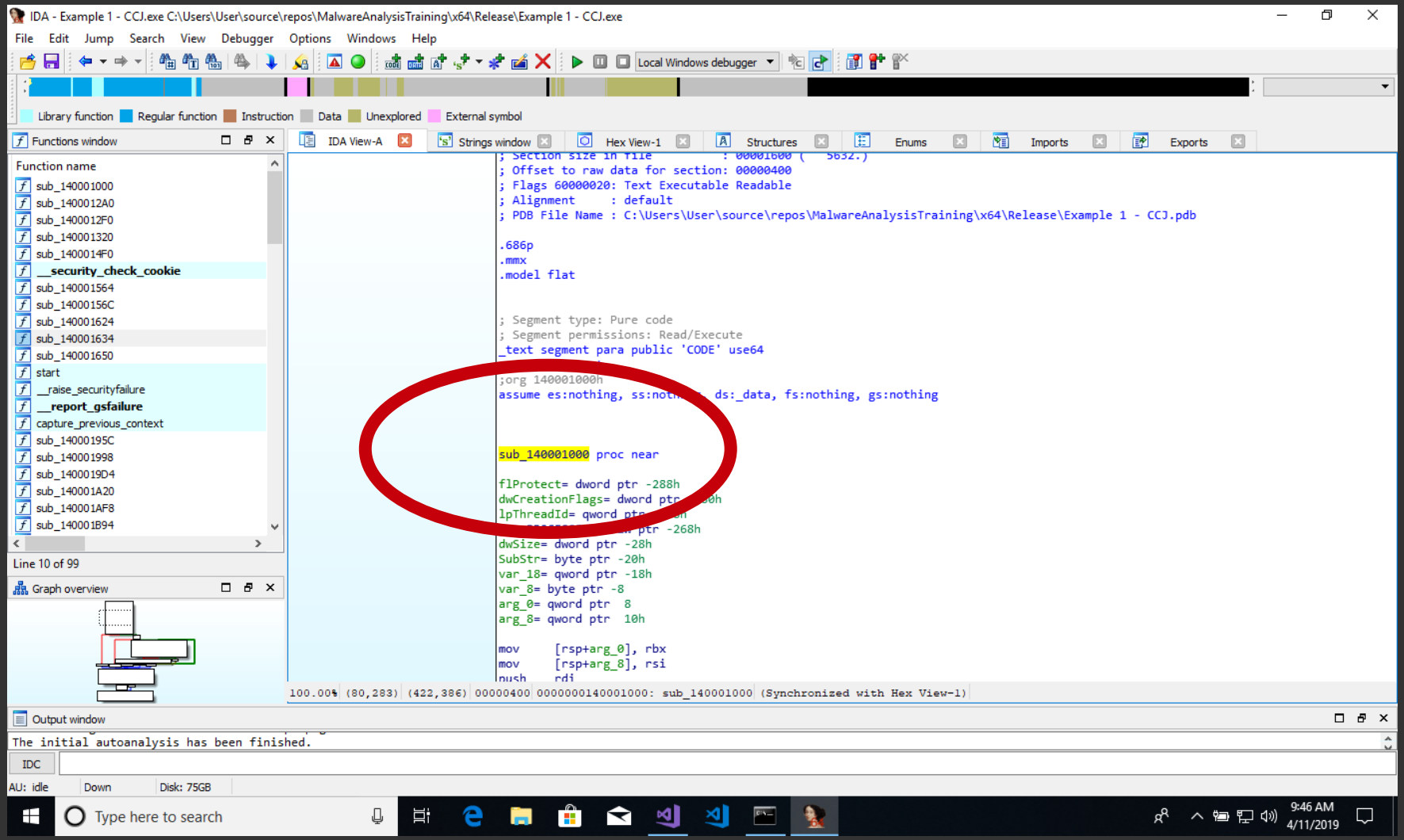
After double clicking the first option we'll be taken to the location of where in the assembly the string is used. I circled the location in red below.

Let's drag ourselves to the top of the assembly and record where we're at.

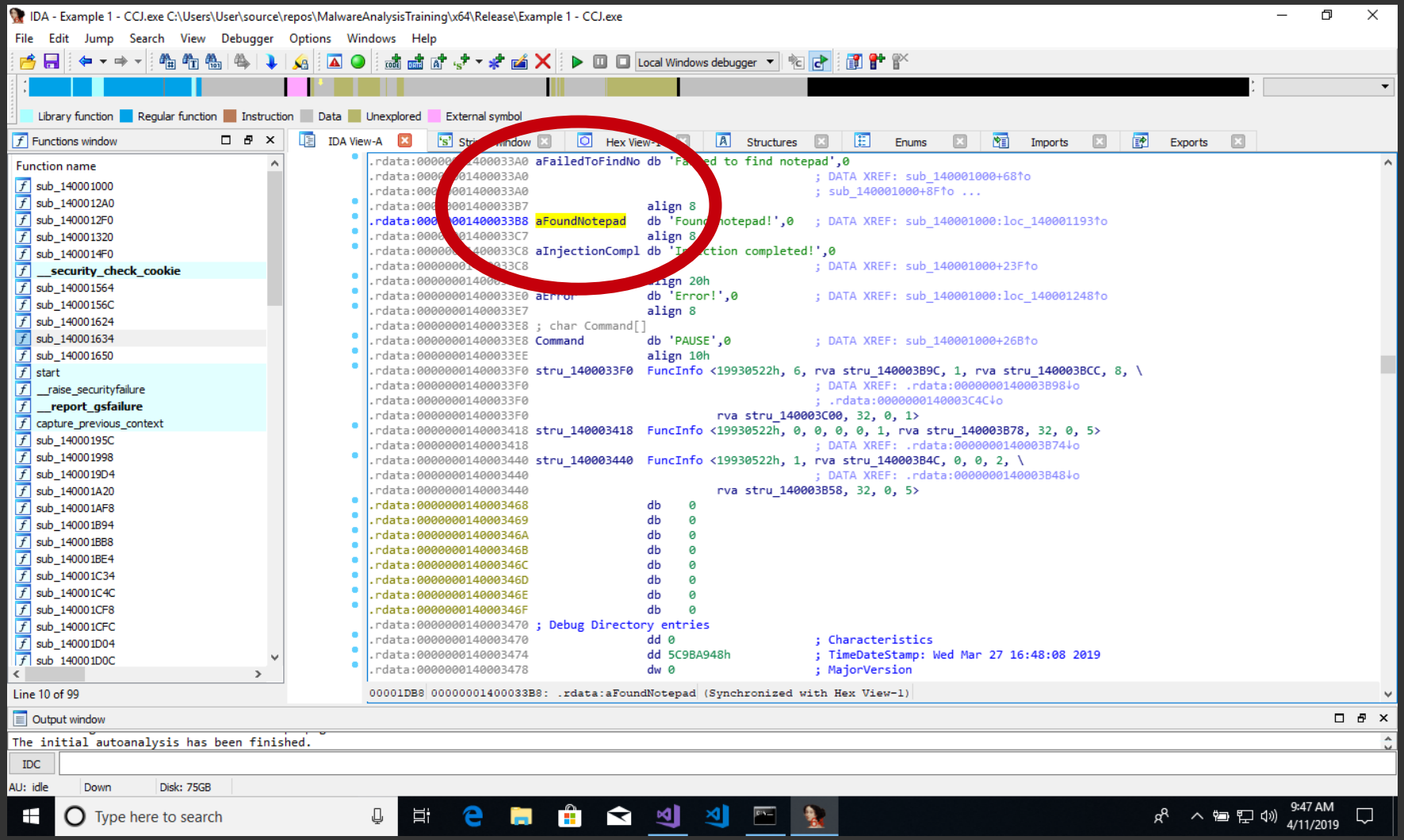


After dragging ourselves to the top block, we can see our name highlighted in yellow below. I also drew a circles around it. IDA tends to name locations with “sub_” and then the offset of where it is in the file.

We'll keep this noted for now in our notes.

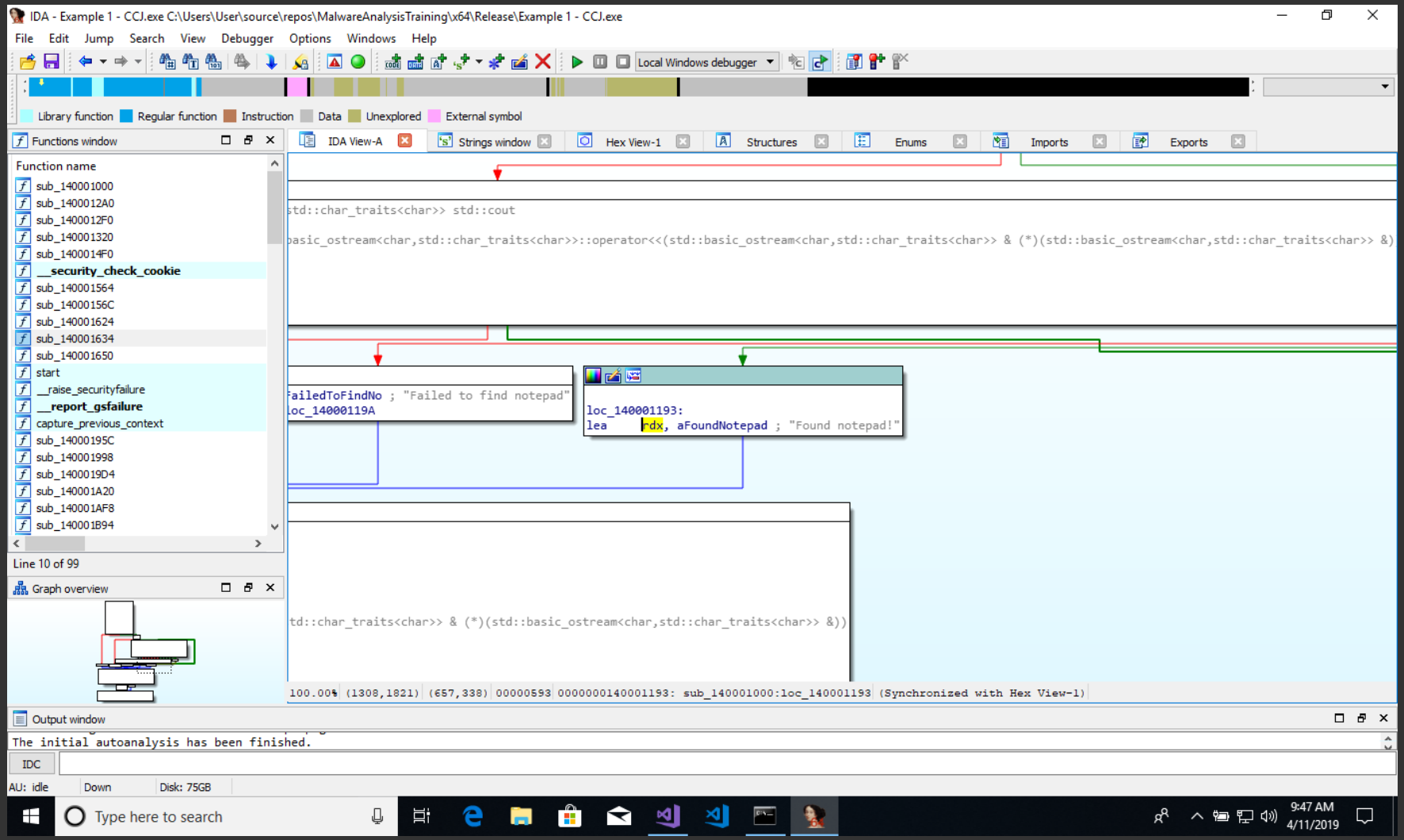


Let's go back to strings and click another string like “Found notepad!”. Remember to hit “X” on the highlighted IDA name below so that the cross references will work.

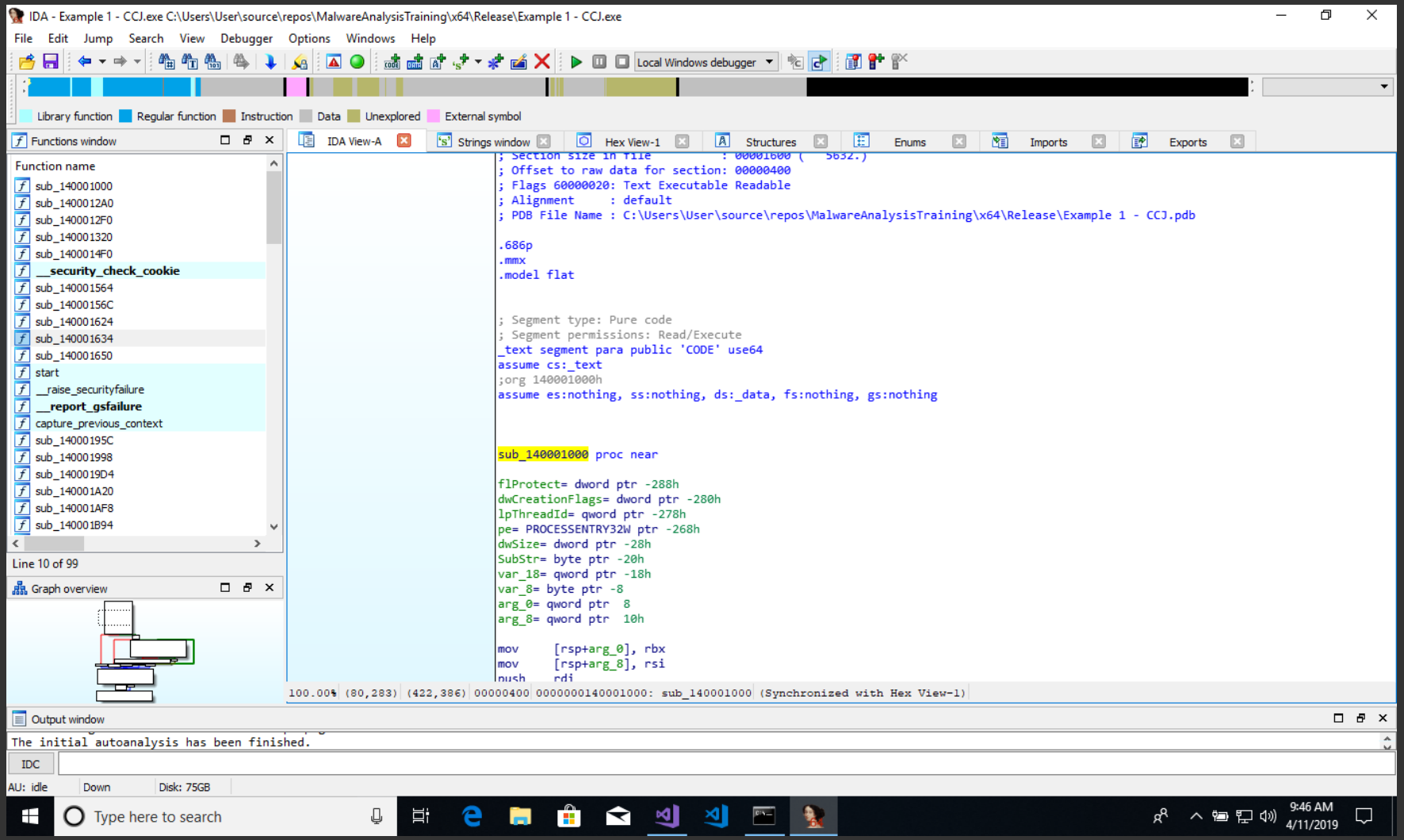


And now we're taken back to the place where “Found notepad!” is used in the assembly.

Let's drag ourselves back up to the top.



Looks like it's within the same place that the failed to find notepad message also resides.



Step 2 - Imports

- In Windows for the programmer to actually interact and perform functionality on the system they must go through the Windows API. Every action performed that creates/modifies/deletes anything on the system must go through this tunnel. This information in most cases, unless its brought in at runtime, is available via the “Imports” section of the binary.

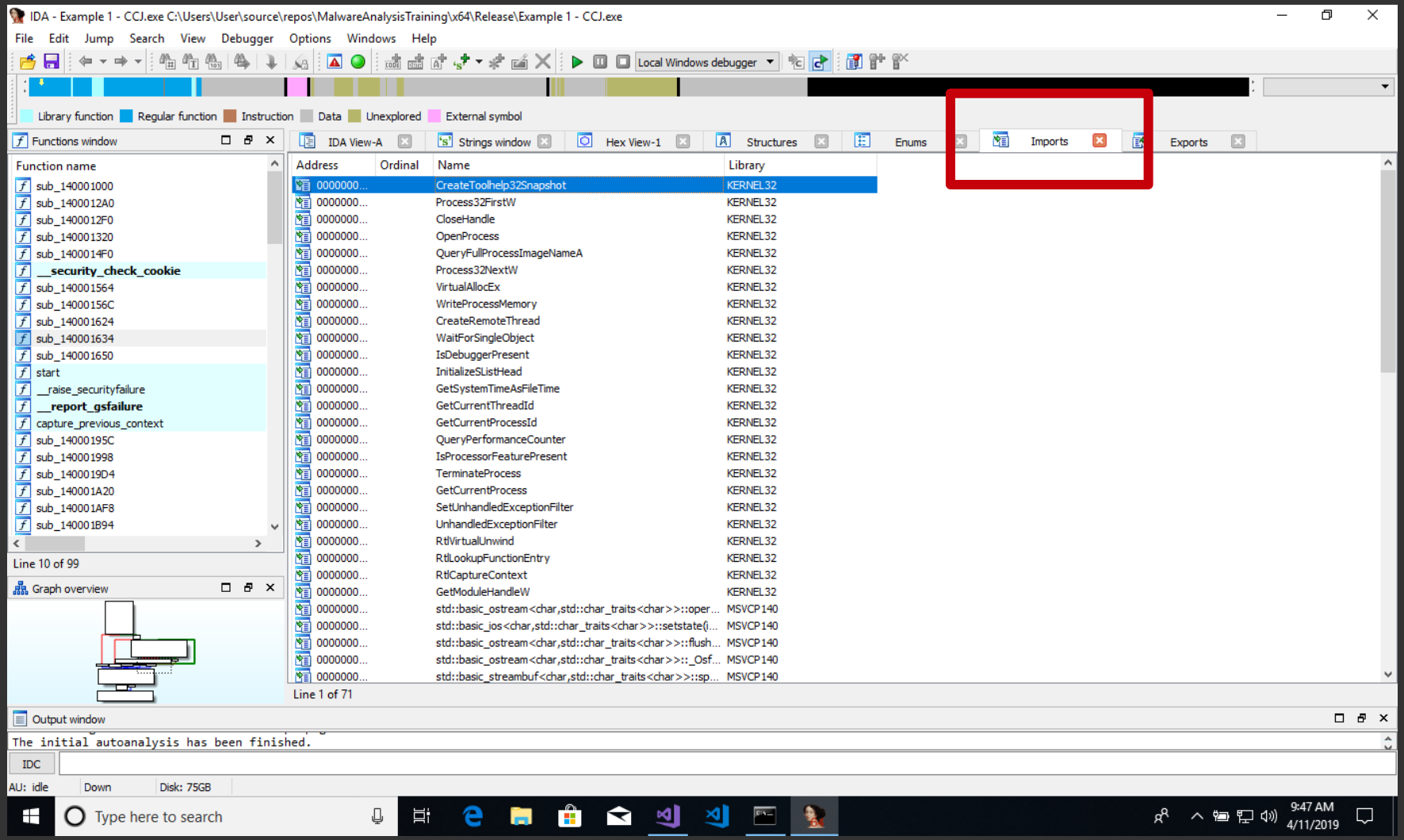
What should we look for?

- Our goal should be when analyzing the imports is to find interesting imports that shine light on the purpose and behavior of the malware.
- Read/Write of memory
- File movement/deletion/creation
- Registry creation/edits/deletion
- The list goes on...

Let's bring up the imports tab and have a look around.

I've highlighted in blue the function I found most interesting at the start.

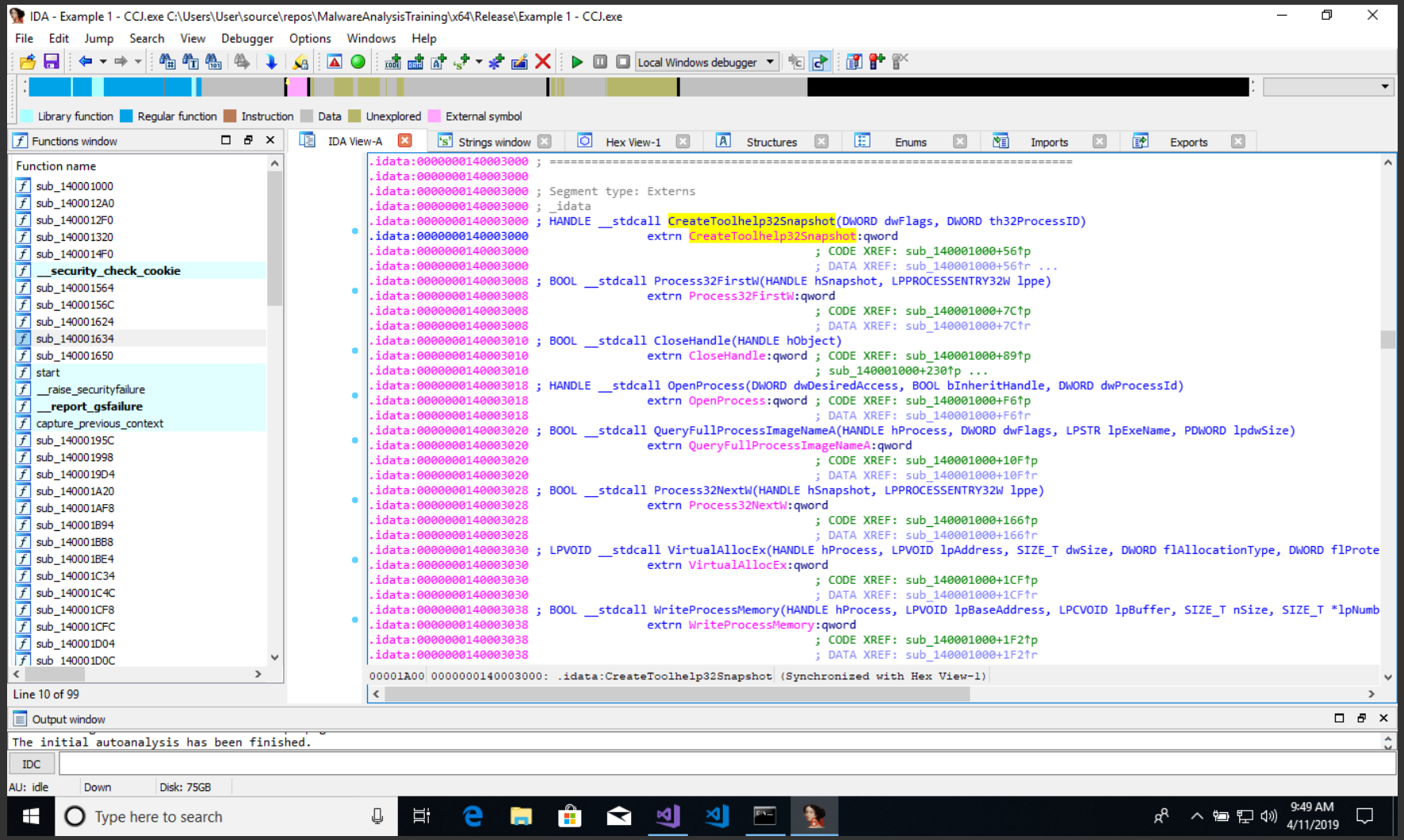
CreateToolhelp32Snapshot



Why CreateToolhelp32Snapshot?

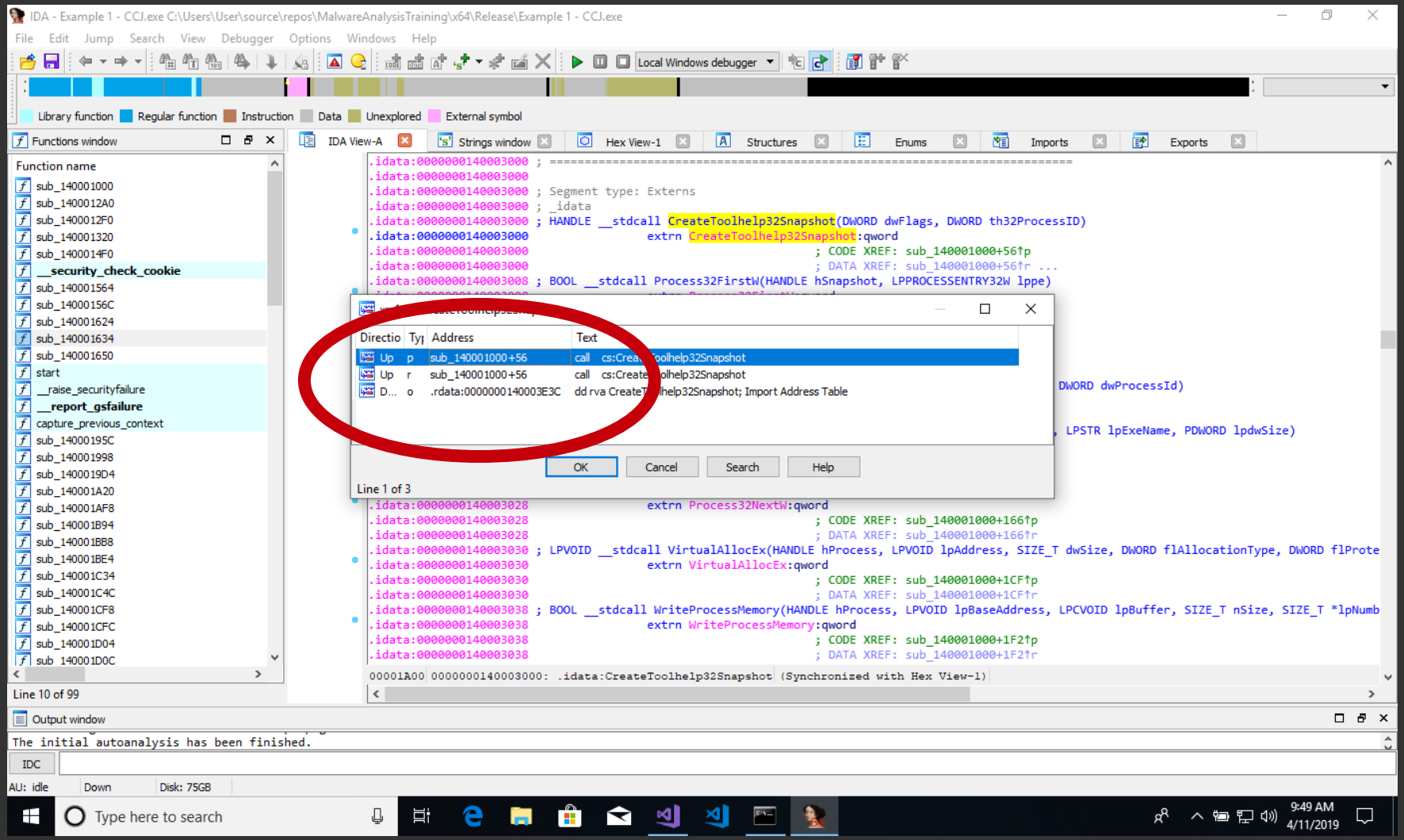
- Allows one to snapshot a list of all processes running on the system.
- We saw at the start when running our binary, a bunch of processes being displayed in our command prompt.
- We can probably deduce that this imported function must be the one allowing the binary to get this information.

Double click the CreateToolhelp32Snapshot and you'll be greeted with this screen. If you press “X” on the pink version of the CreateToolhelp32Snapshot you'll bring up the cross references for this imported function in the assembly.



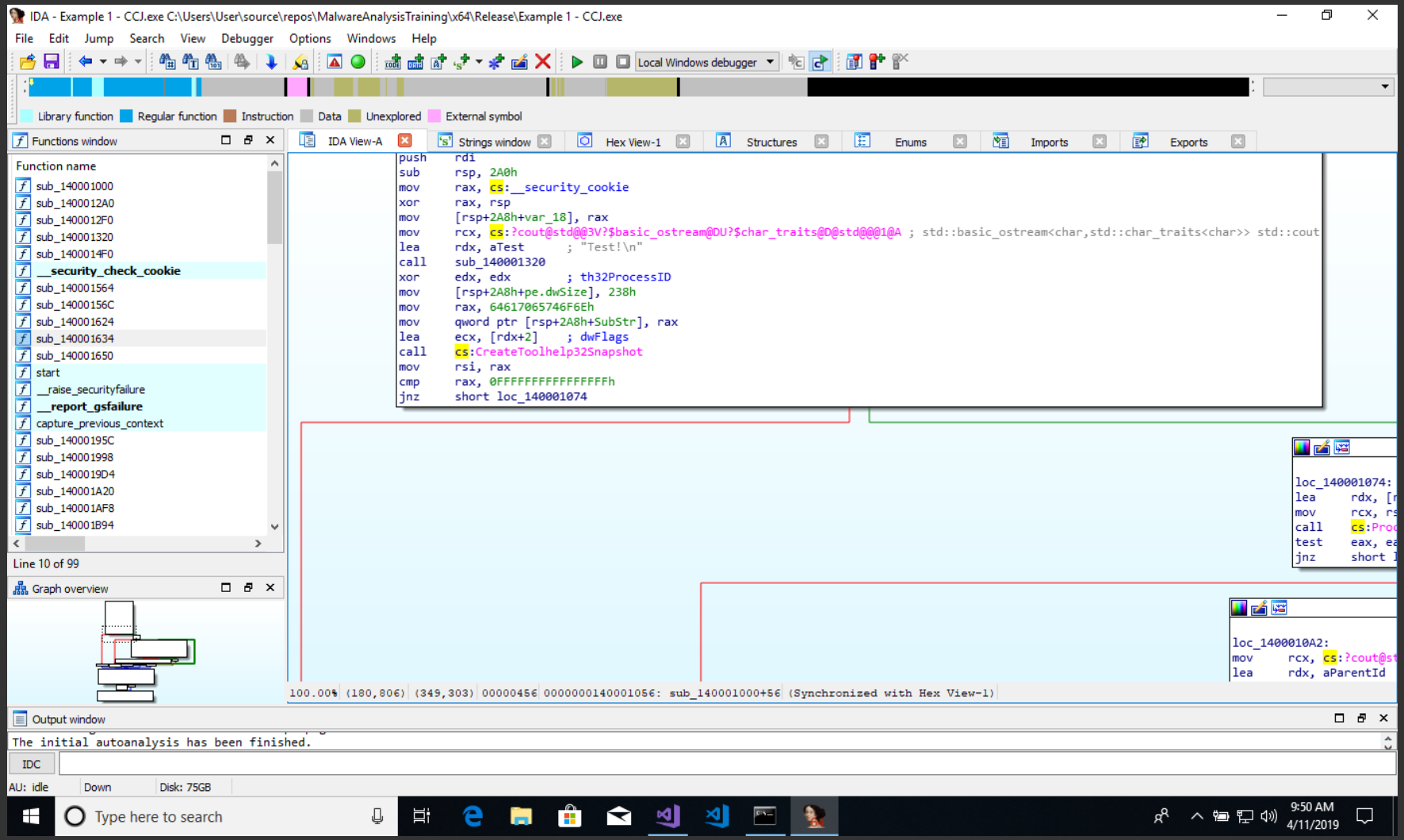
Looking at the cross references (xrefs). We can see where it's located in the assembly.

Circles below is also where the address is relative to the IDA name for the function. Recognize this place? You should, Failed to find notepad is there.



Let's click the first spot and be greeted with the assembly block of where CreateToolhelp32Snapshot is in.

At this point let's start diving into dynamic analysis.



Exports

- We'll discuss exports in a later example, for now know that exports are pretty important for getting a starting point in some cases.

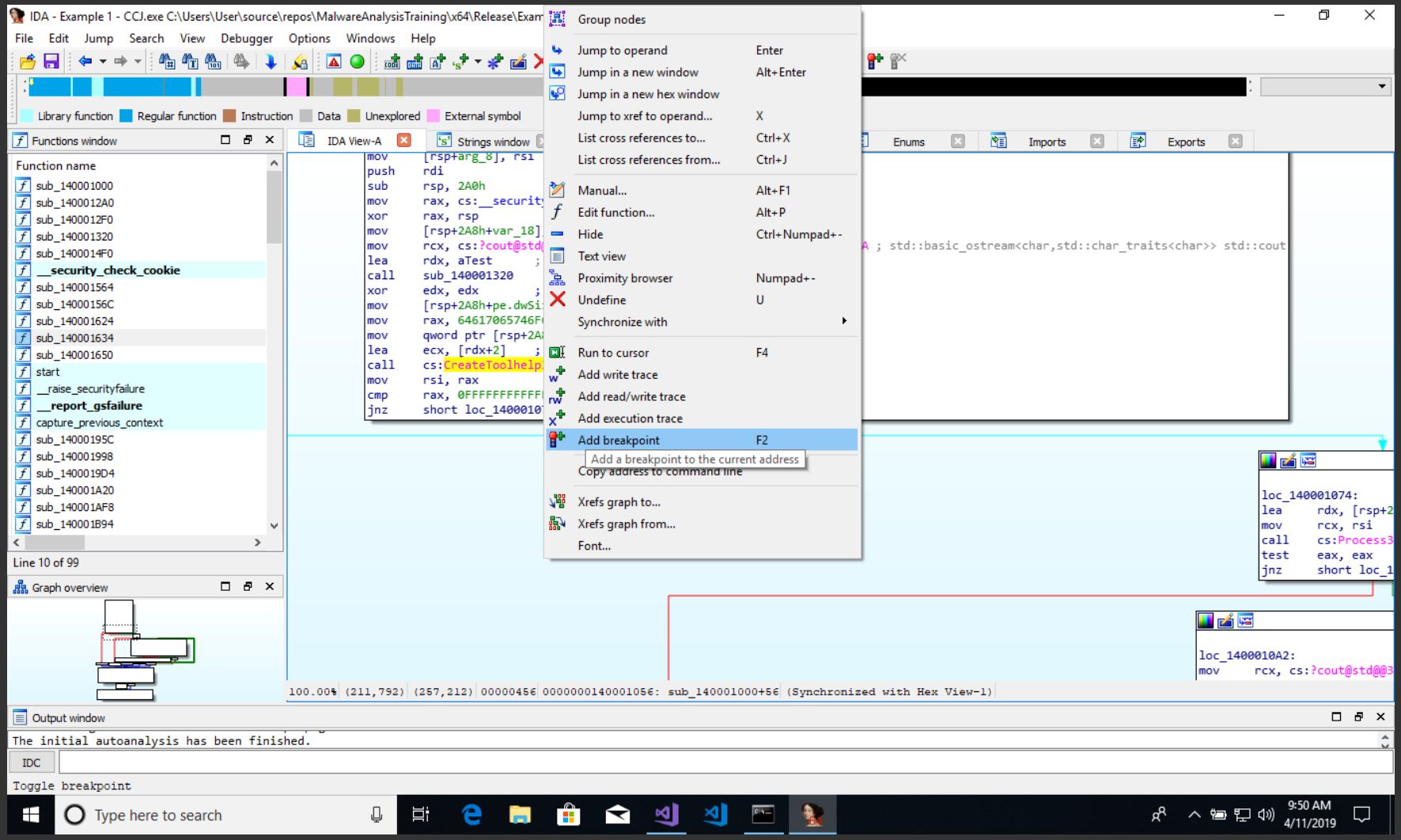
Dynamic Analysis

- Allows one to inspect data, memory, and functions at runtime.
- Using breakpoints and debugging the malware we can come up with faster assumptions while we run it.

When to use Dynamic Analysis?

- When the malware/binary becomes to difficult to figure out from static analysis alone.
- Assembly takes quite some time to drudge through and dynamic analysis can quickly find answers that static analysis may not have immediately.
- The only negative is that you have to run the malware (Better hope there's no VM escape in the malware)

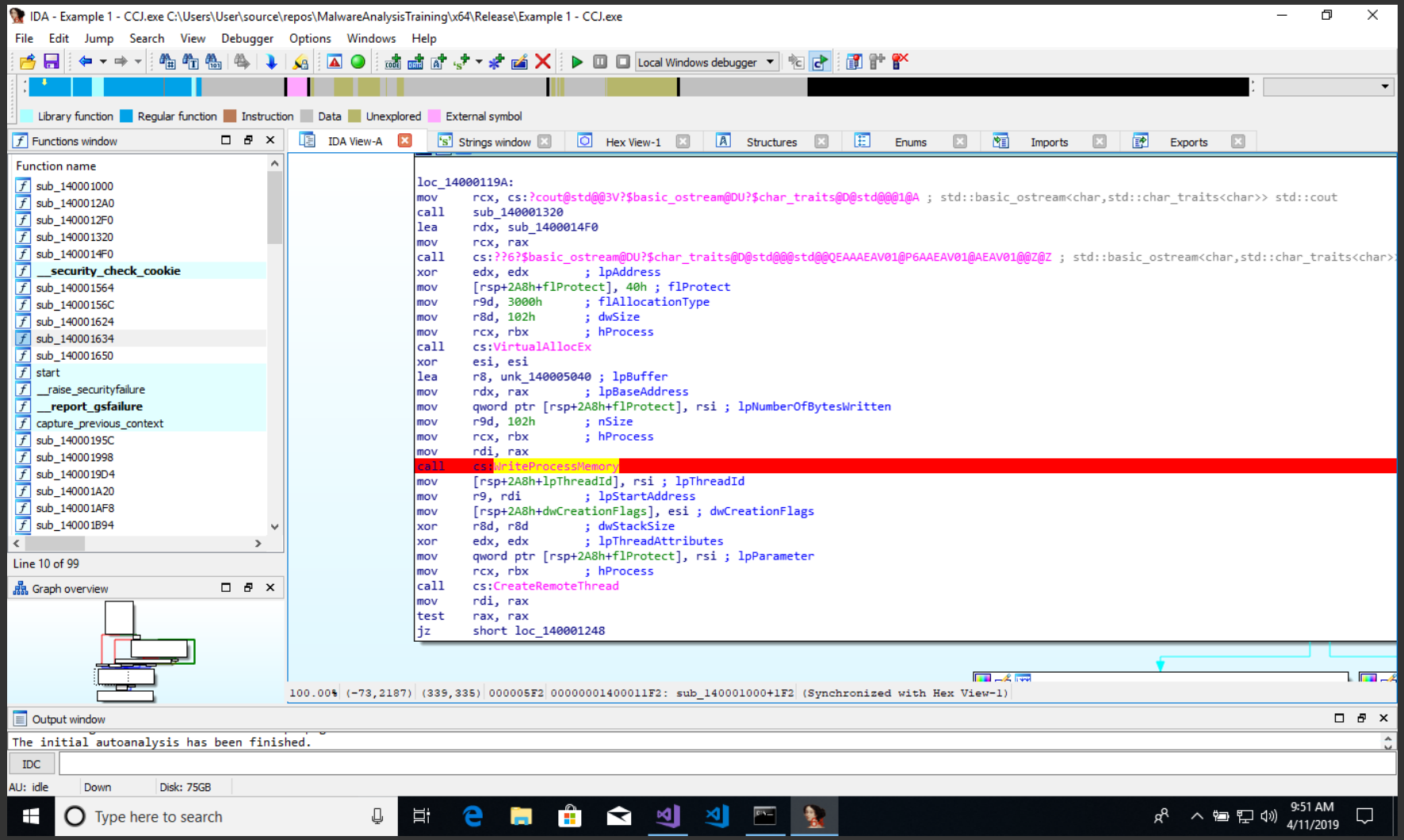
Let's create a breakpoint on “CreateToolhelp32Snapshot”. Do this by clicking the line that the pink text of the function is on and right click and hover down to the “Add breakpoint” option.



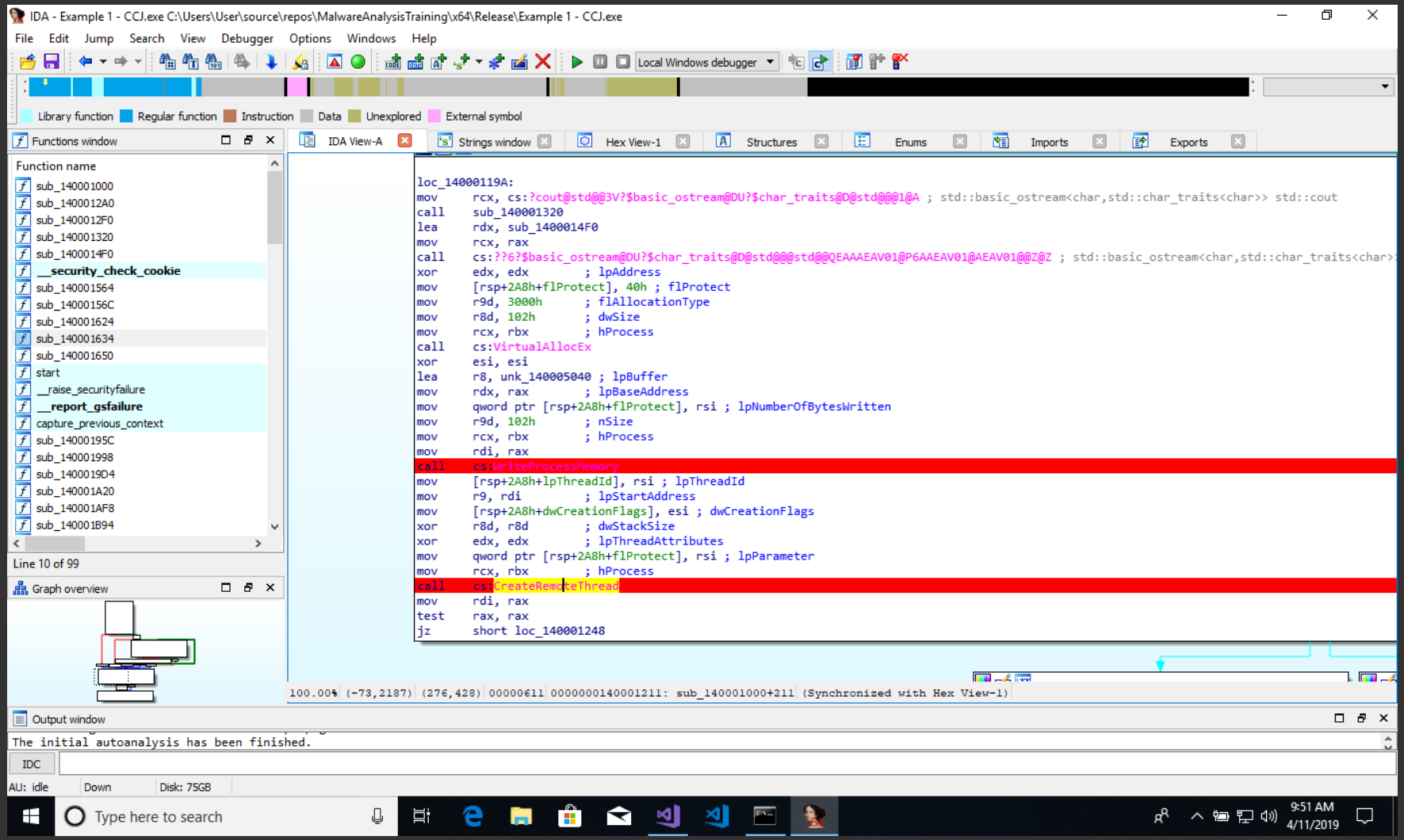
Go back to the “Imports” tab and do this for a few other functions like

WriteProcessMemory

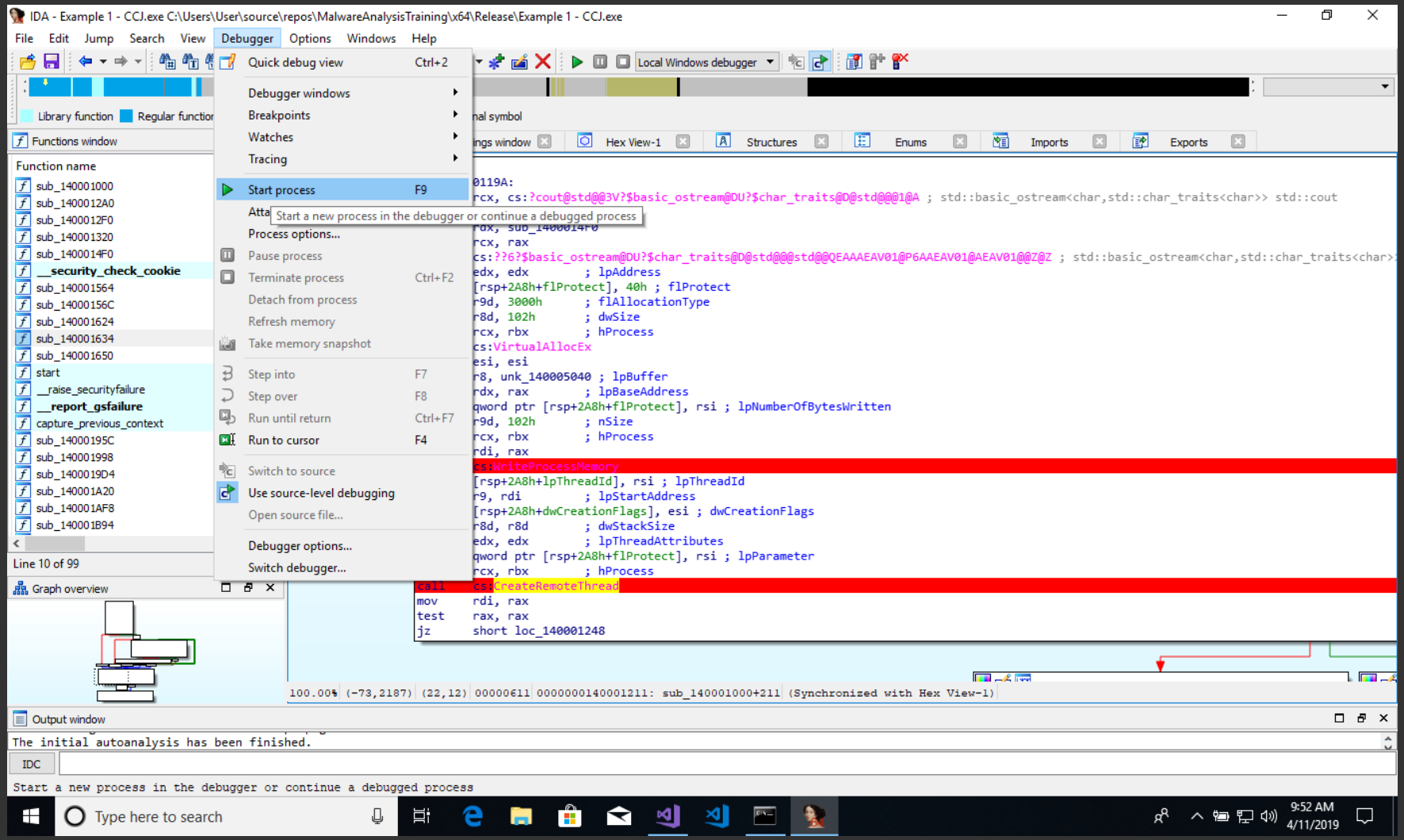
CreateRemoteThread



These are some pretty important functions. So just trust me on breakpointing them for now.

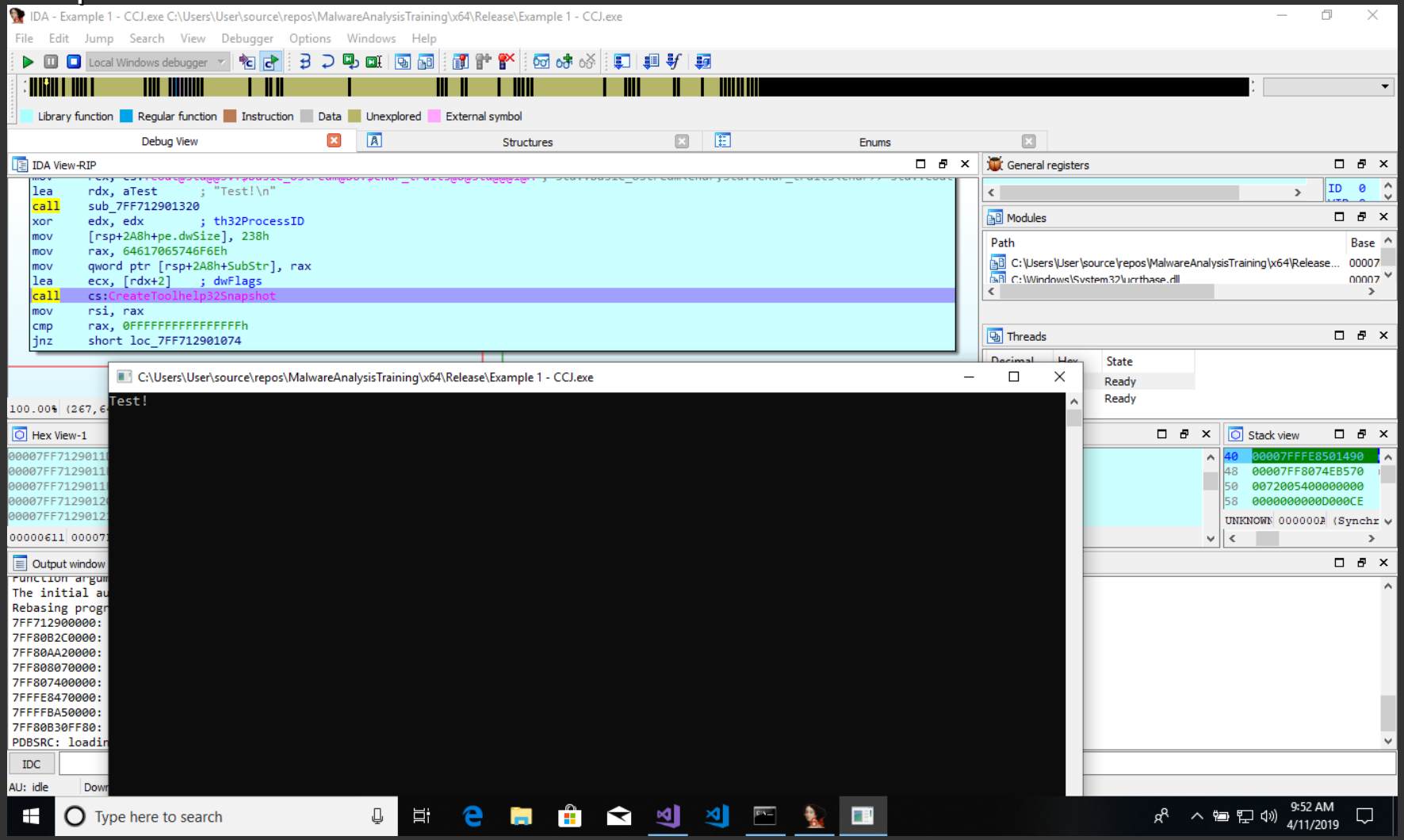


Let's go ahead and now start up the malware again in IDA.

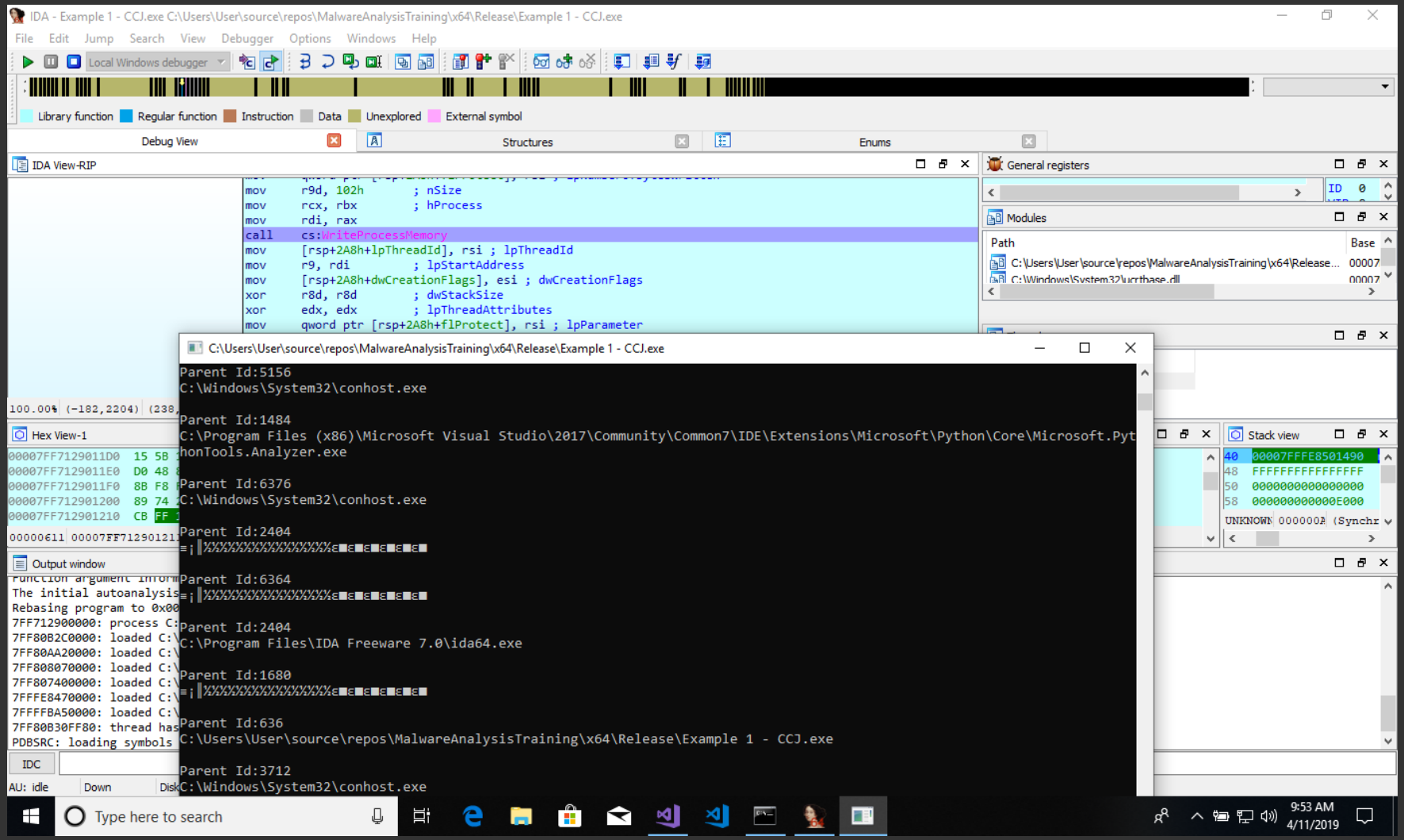


We notice that we stopped at CreateToolhelp32Snapshot because it is now colored purple.

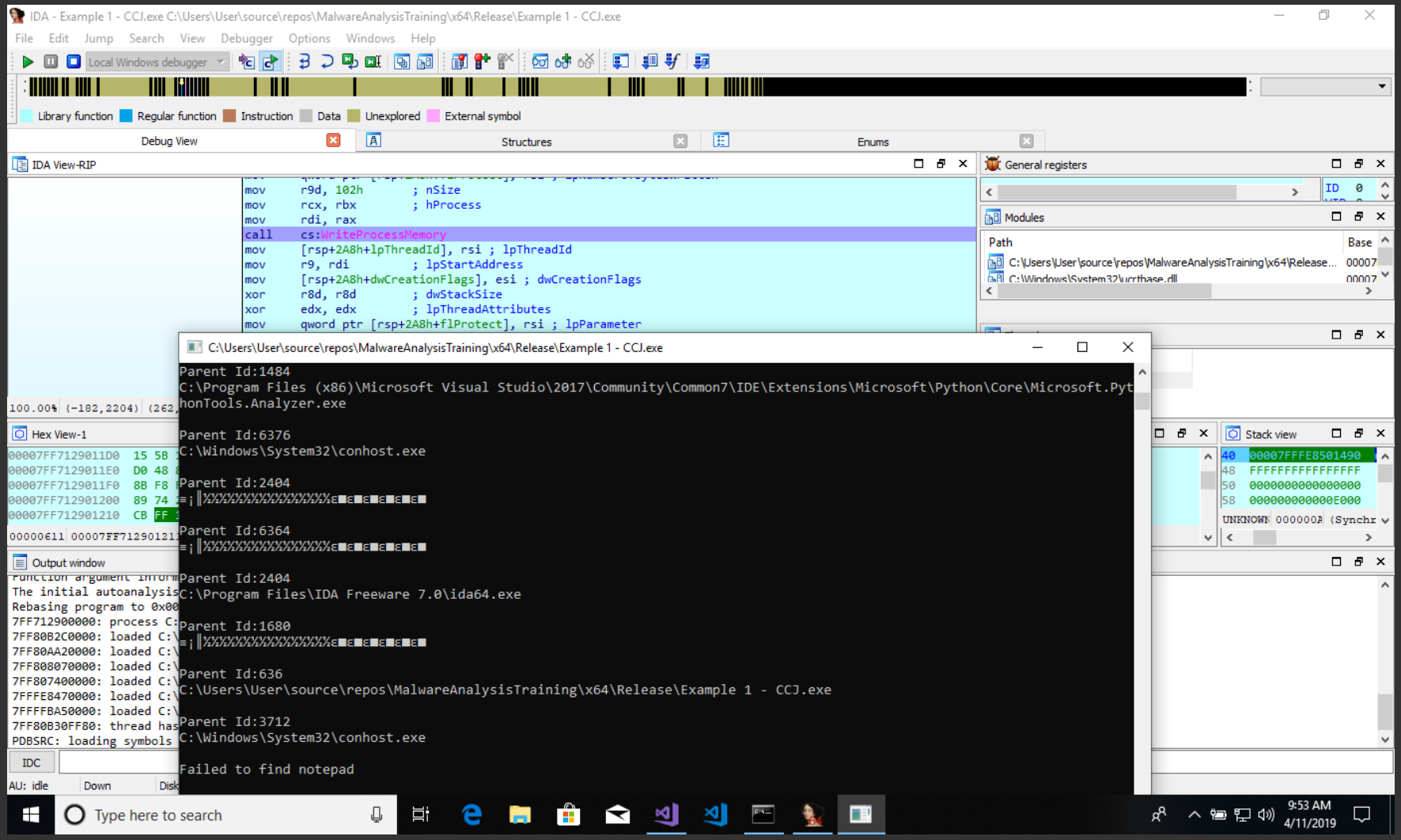
Pulling the command prompt up for the malware shows us it has only printed “Test!” so far. This means the process output doesn't happen till after the snapshot.



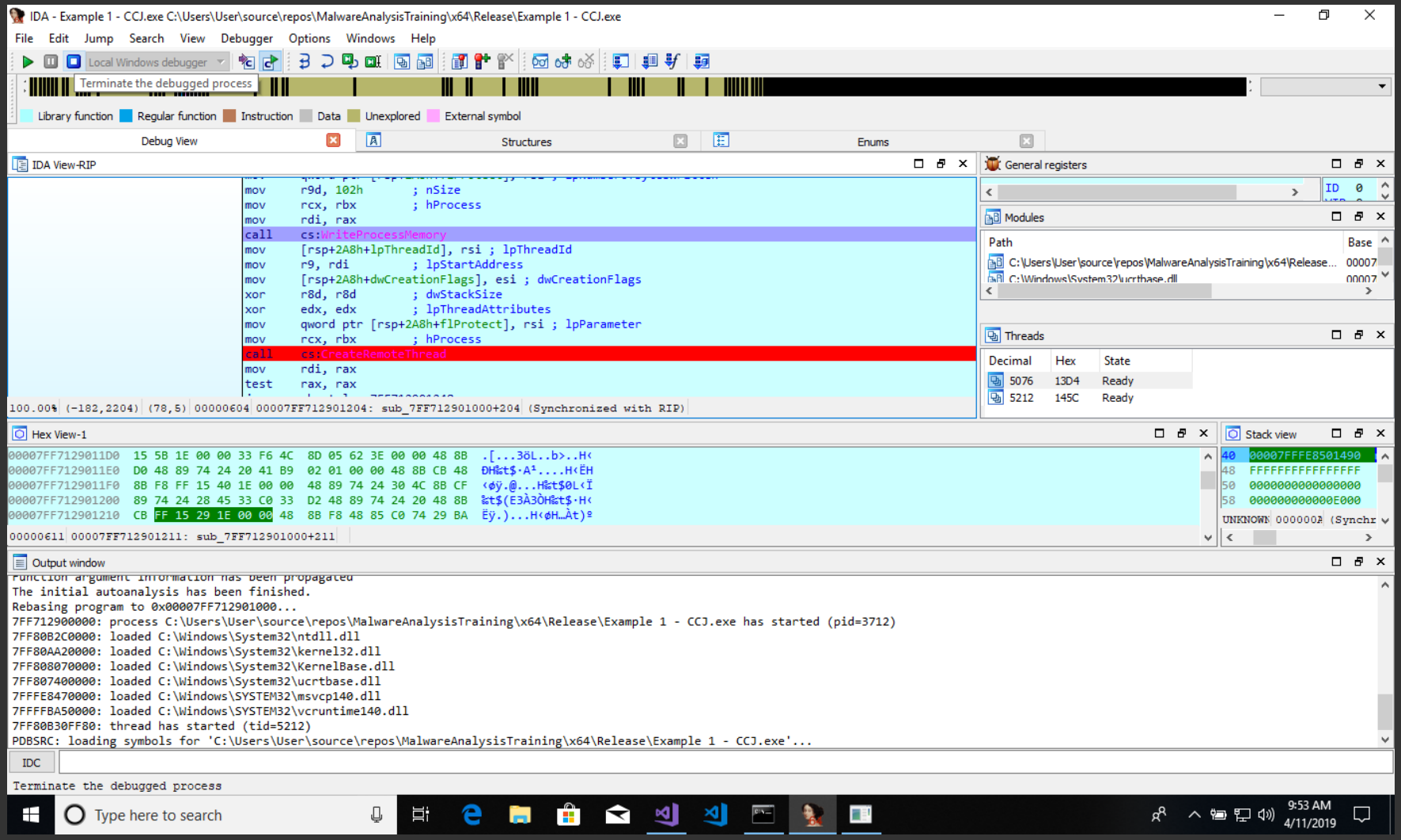
Hitting the play button in the top left. The next stopping point should be “WriteProcessMemory”. Now we see all the output of the processes again.



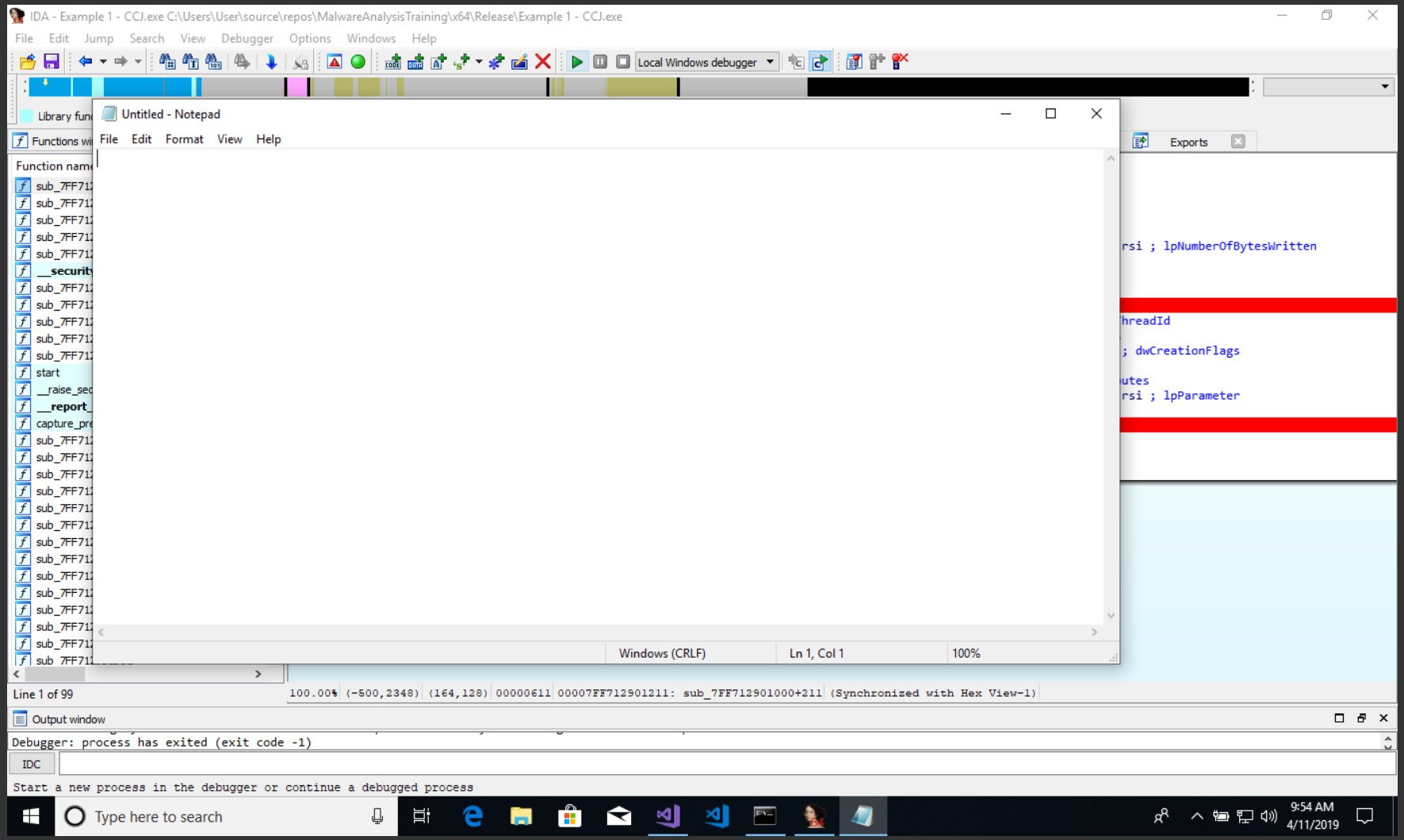
We'll scroll down the prompt a bit and see “Failed to find notepad” again. So long story short we've confirmed in the timeline that the Snapshot is used for listing the processes and the hypothesis now is that it must be looking for a notepad process.



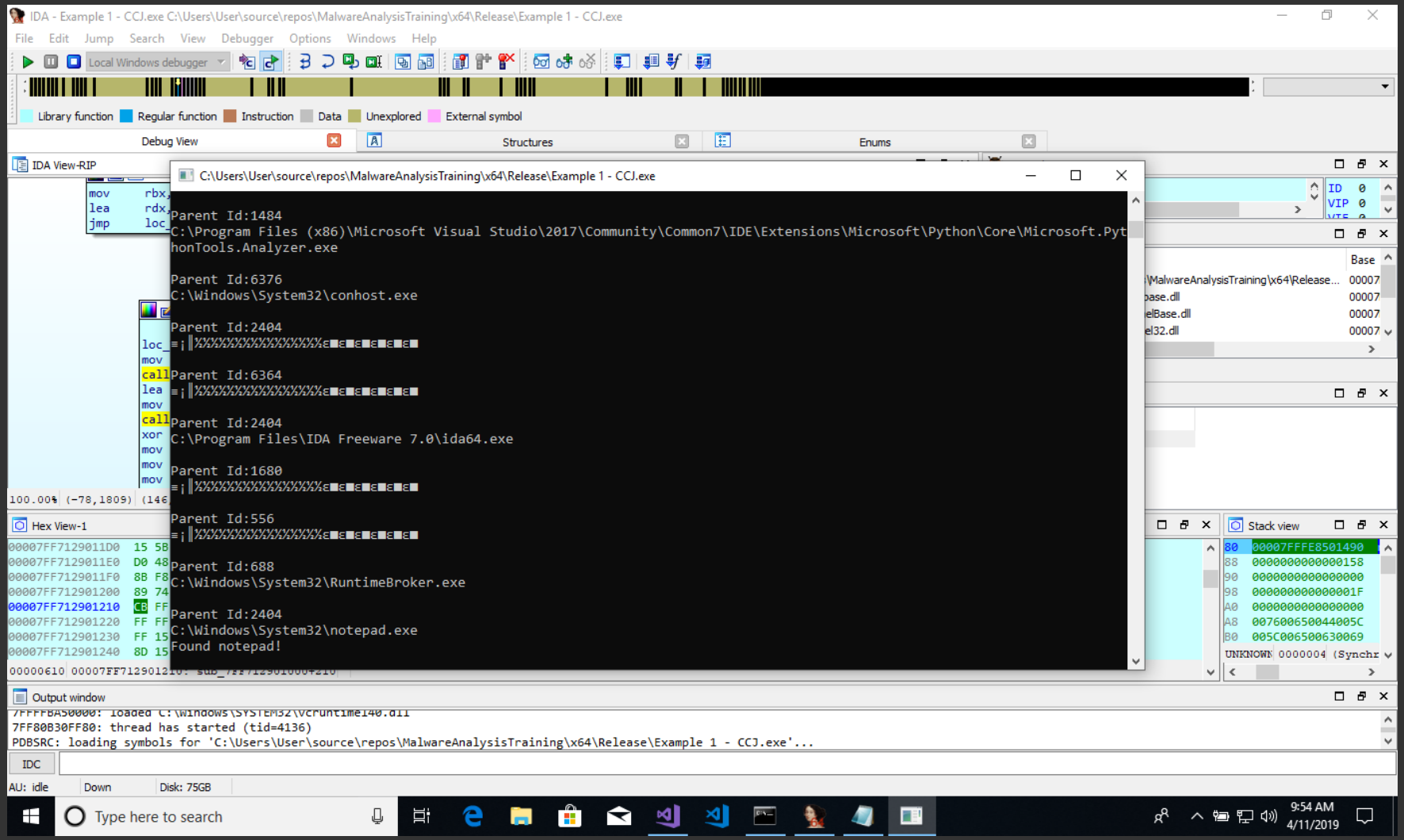
Lets go ahead and restart the malware in IDA.



This time lets open a notepad and watch what happens to confirm a few things.



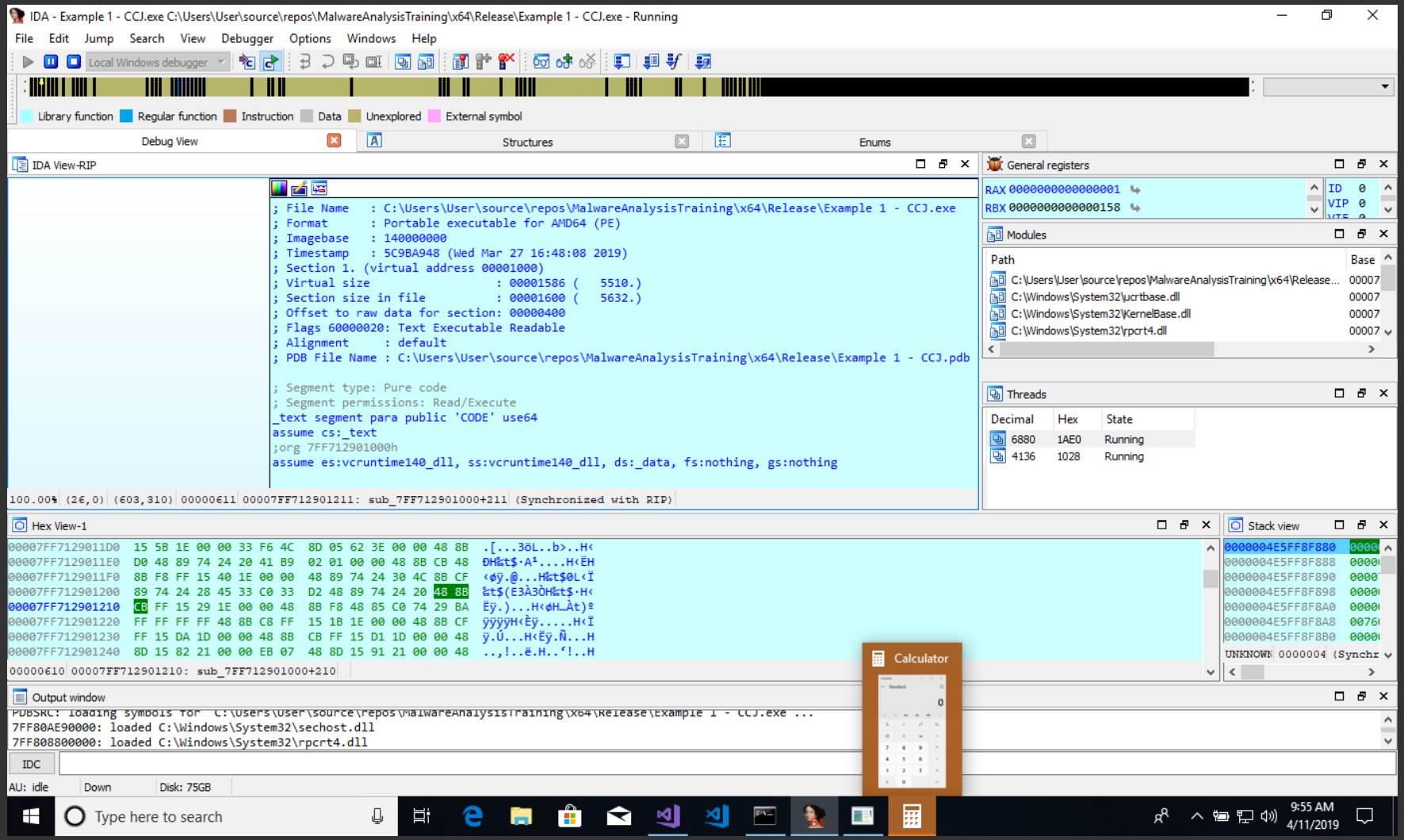
Running it again until we hit “WriteProcessMemory” shows us the “Found notepad!” message. So our hypothesis is shown true. It is looking for a notepad.



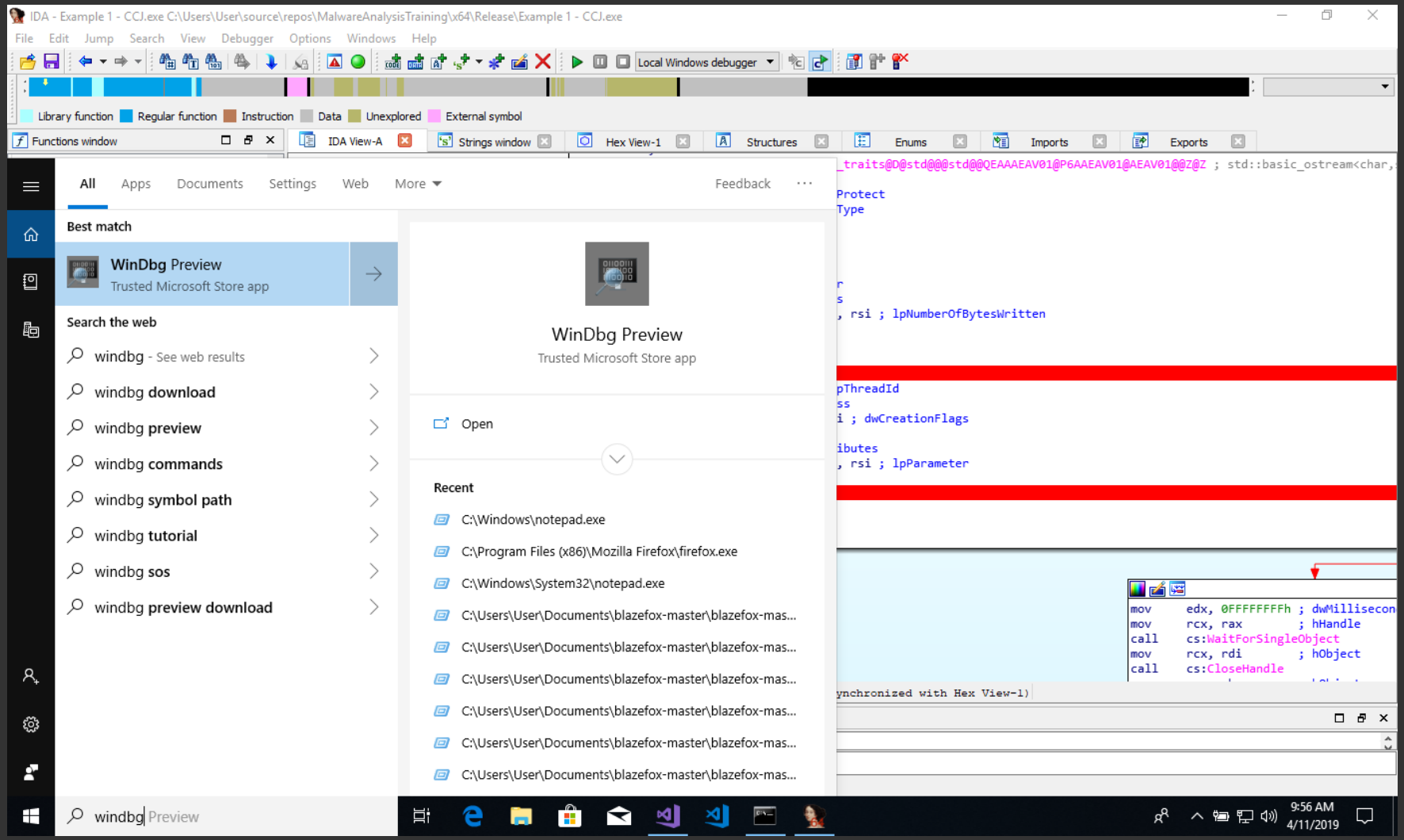
Now hit play until no more breakpoints are hit and watch what happens.

The calculator pops up, but we notice that the notepad vanishes along with it.

Time to get to the bottom of this.

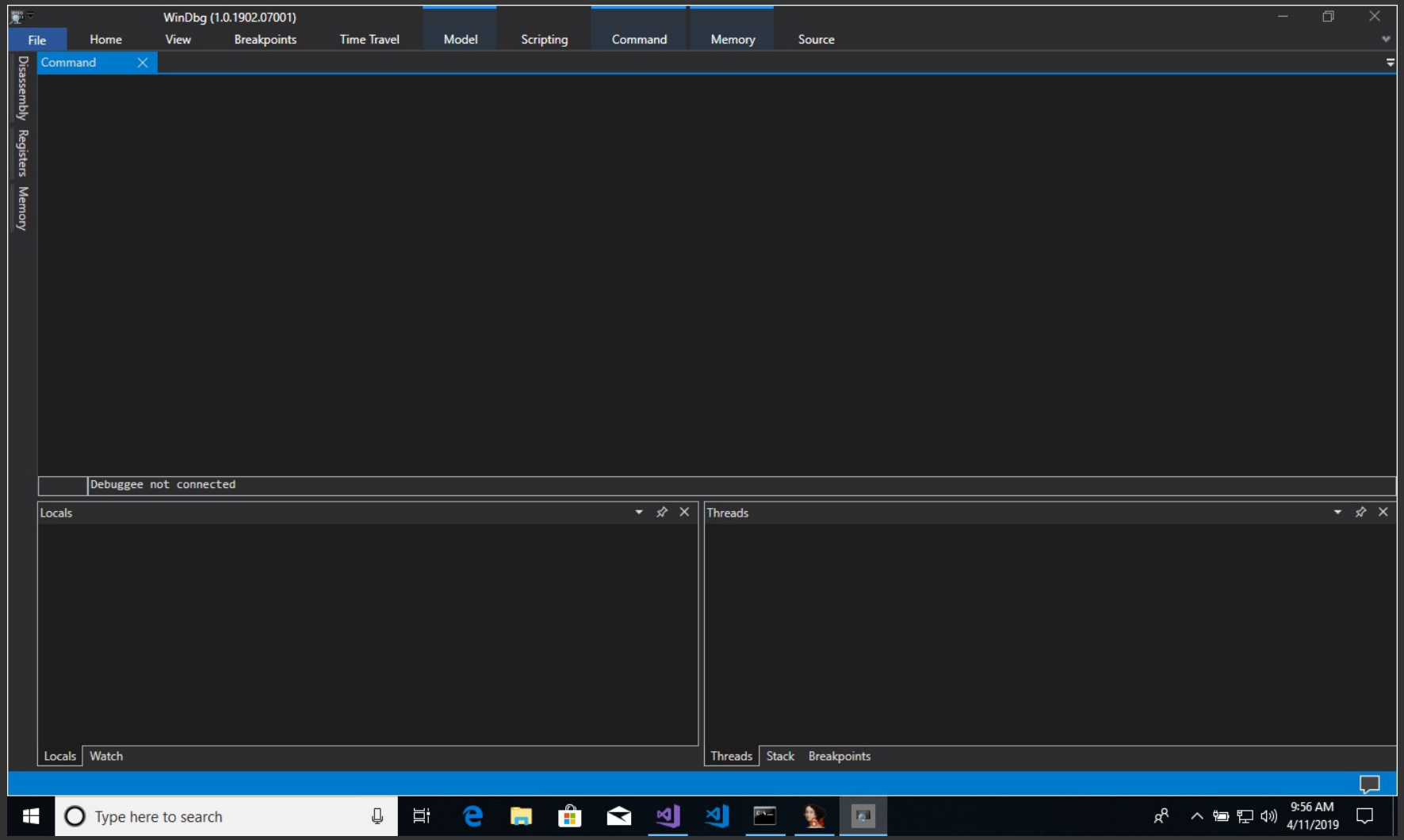


Time to open windbg.

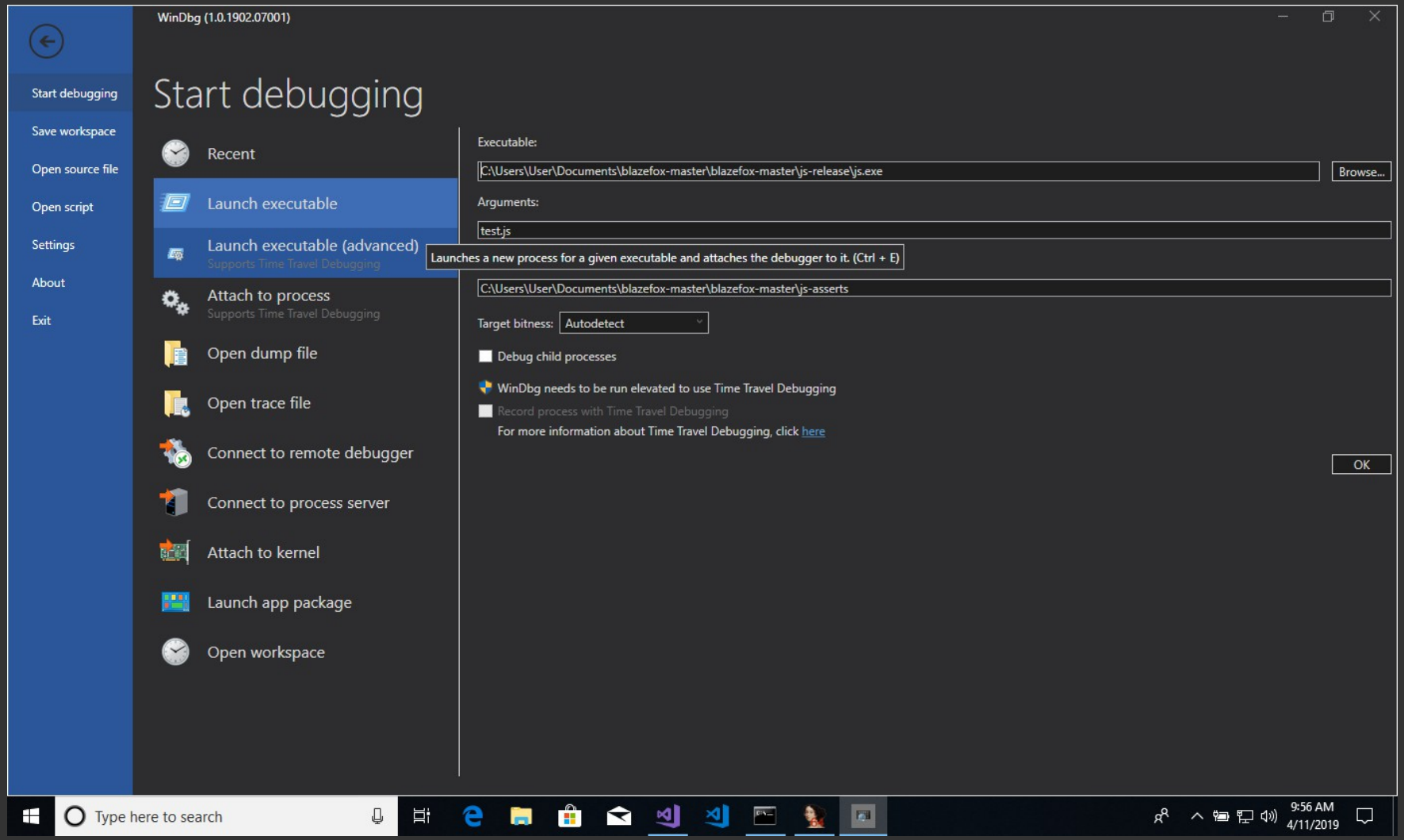


Once windbg is opened it should look like this in some way.

I'm currently using the dark skin variant.

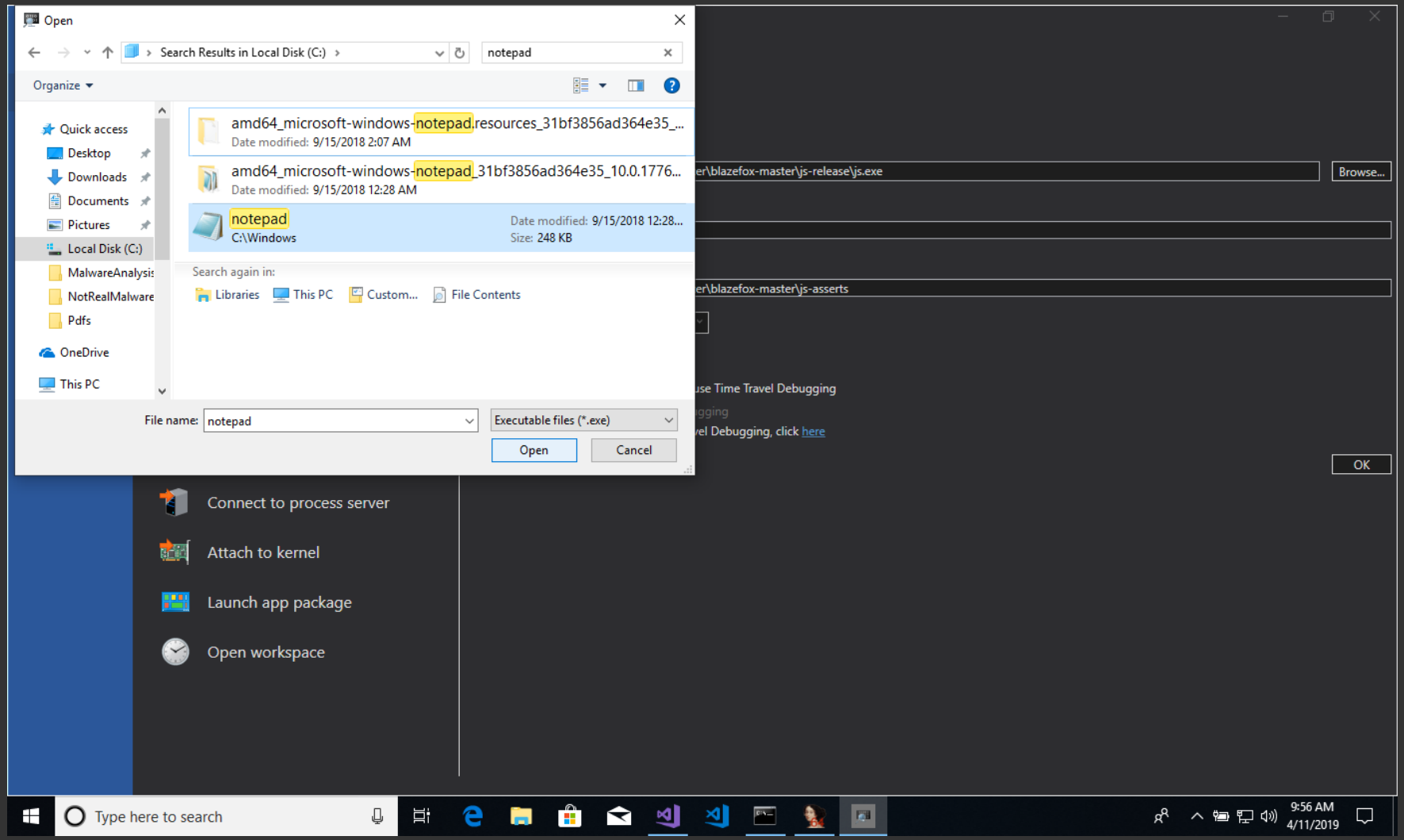


Click File in the top left and let's click “Launch executable”.



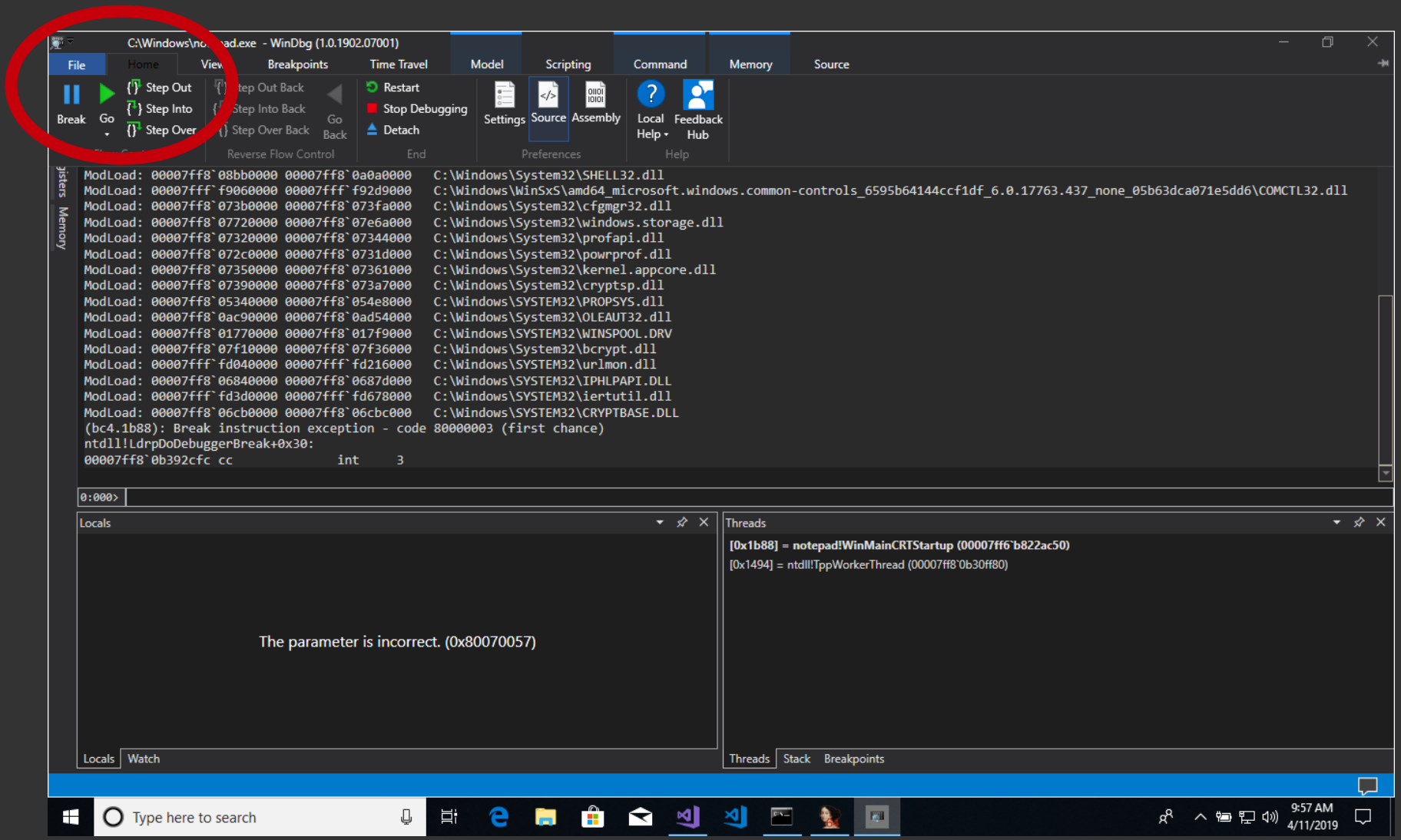
Open notepad.

Notepad tends to reside in system32, but if you click the Local Disk C: and type notepad in the search, notepad should pop up pretty fast.

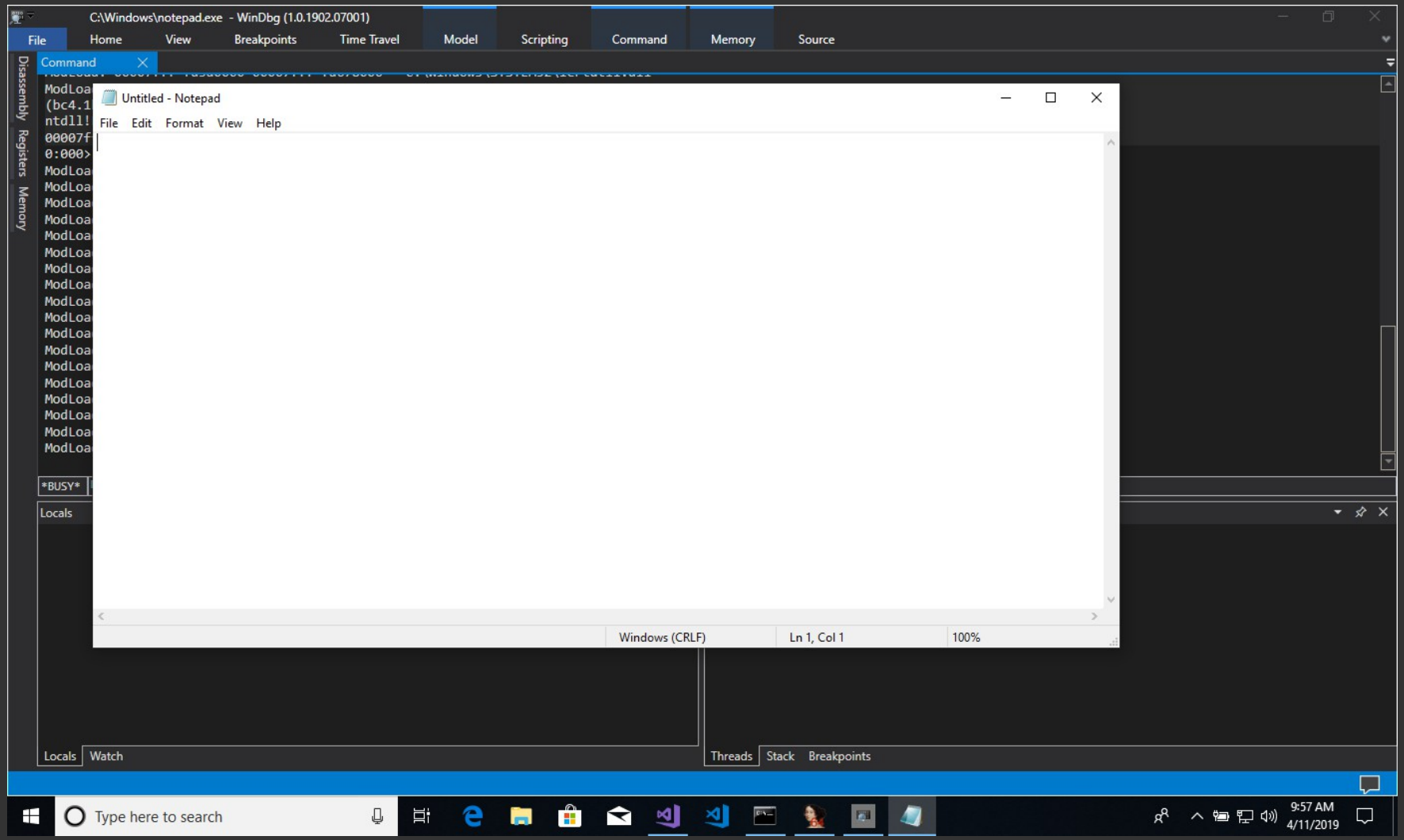


After hitting open, we should see a screen like this and windbg should now be watching notepad.

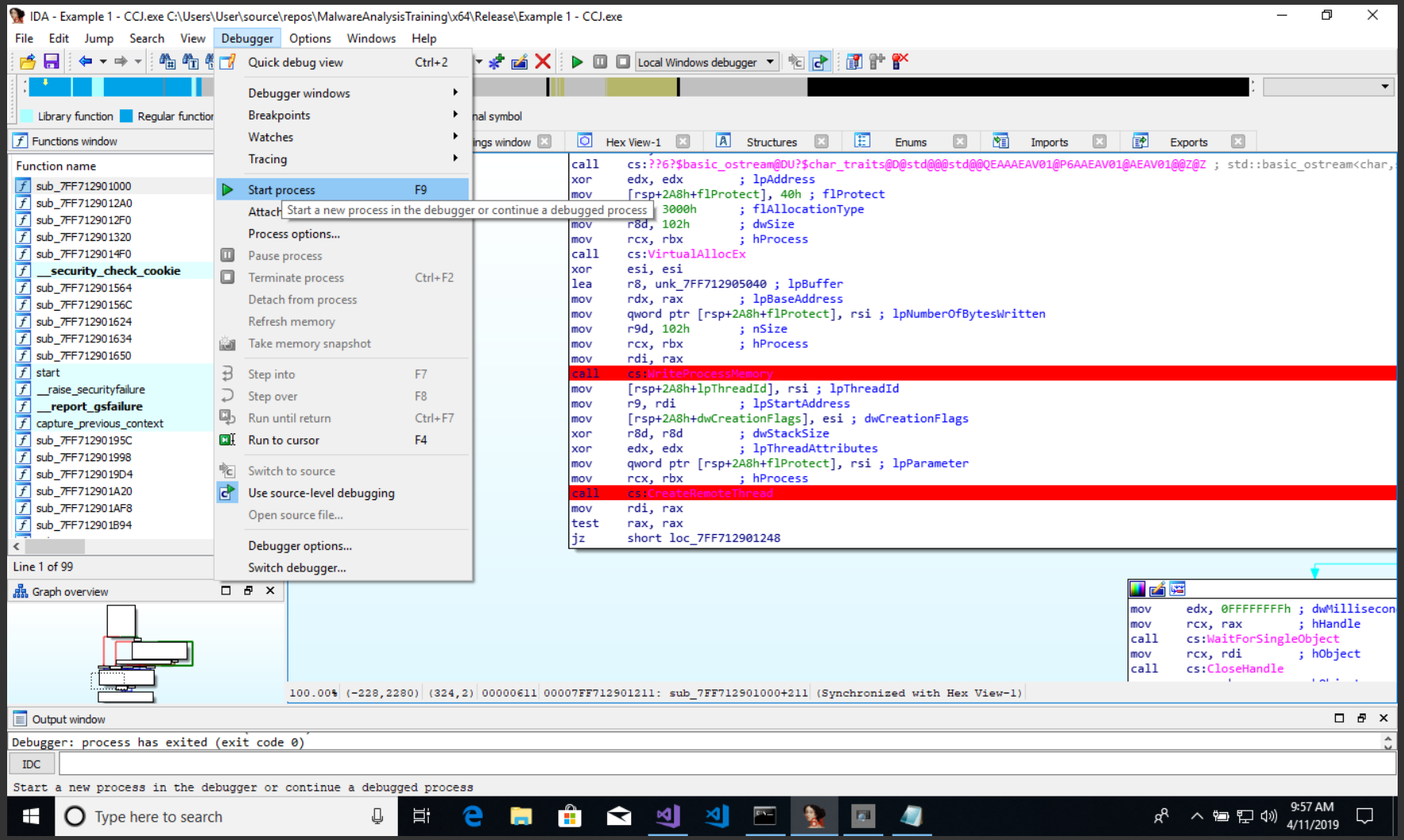
Hitting the giant “Go” play button to start notepad.



We should now see our notepad start running and windbg running in the background.

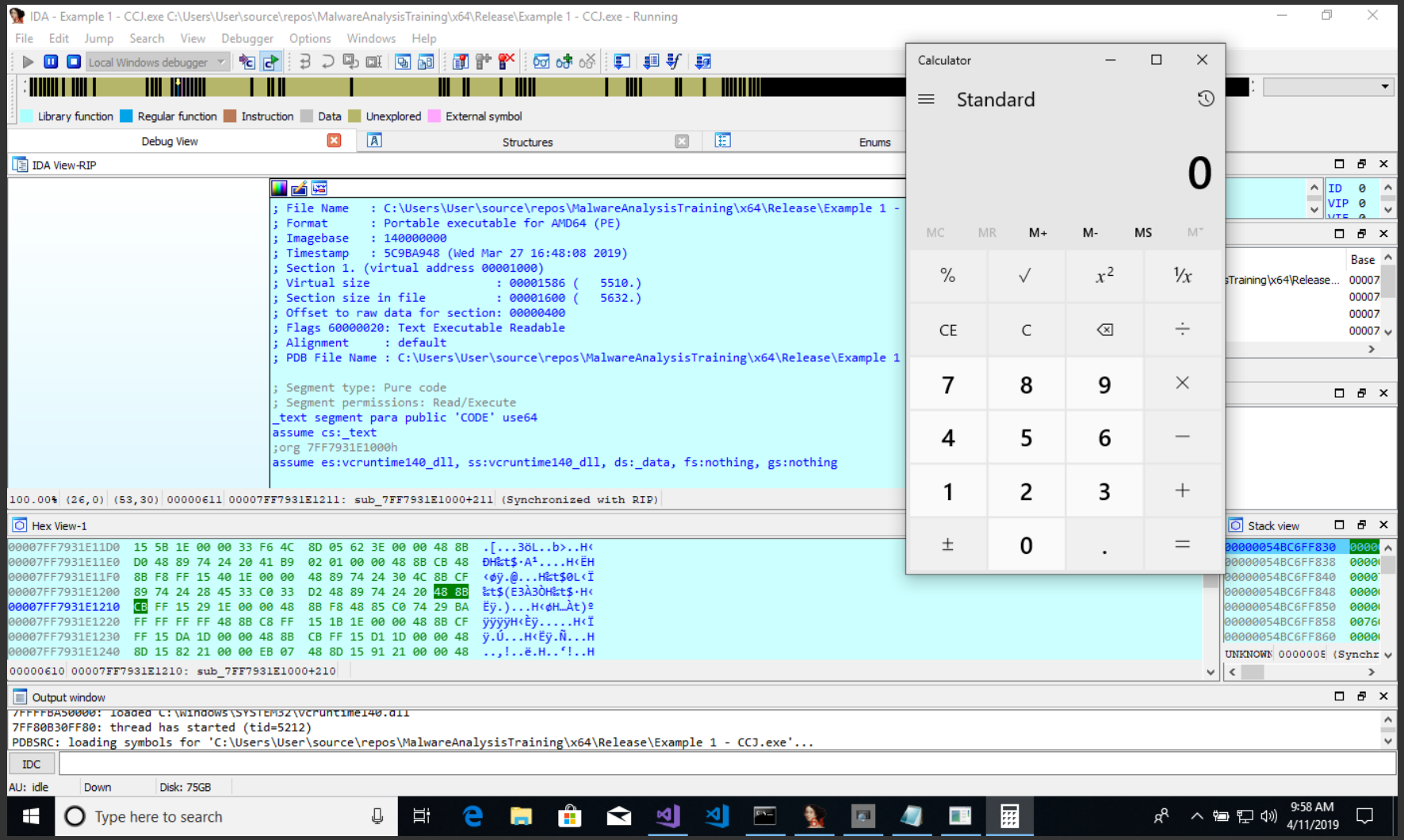


Head back into IDA and let us start the malware process over again and watch what happens.



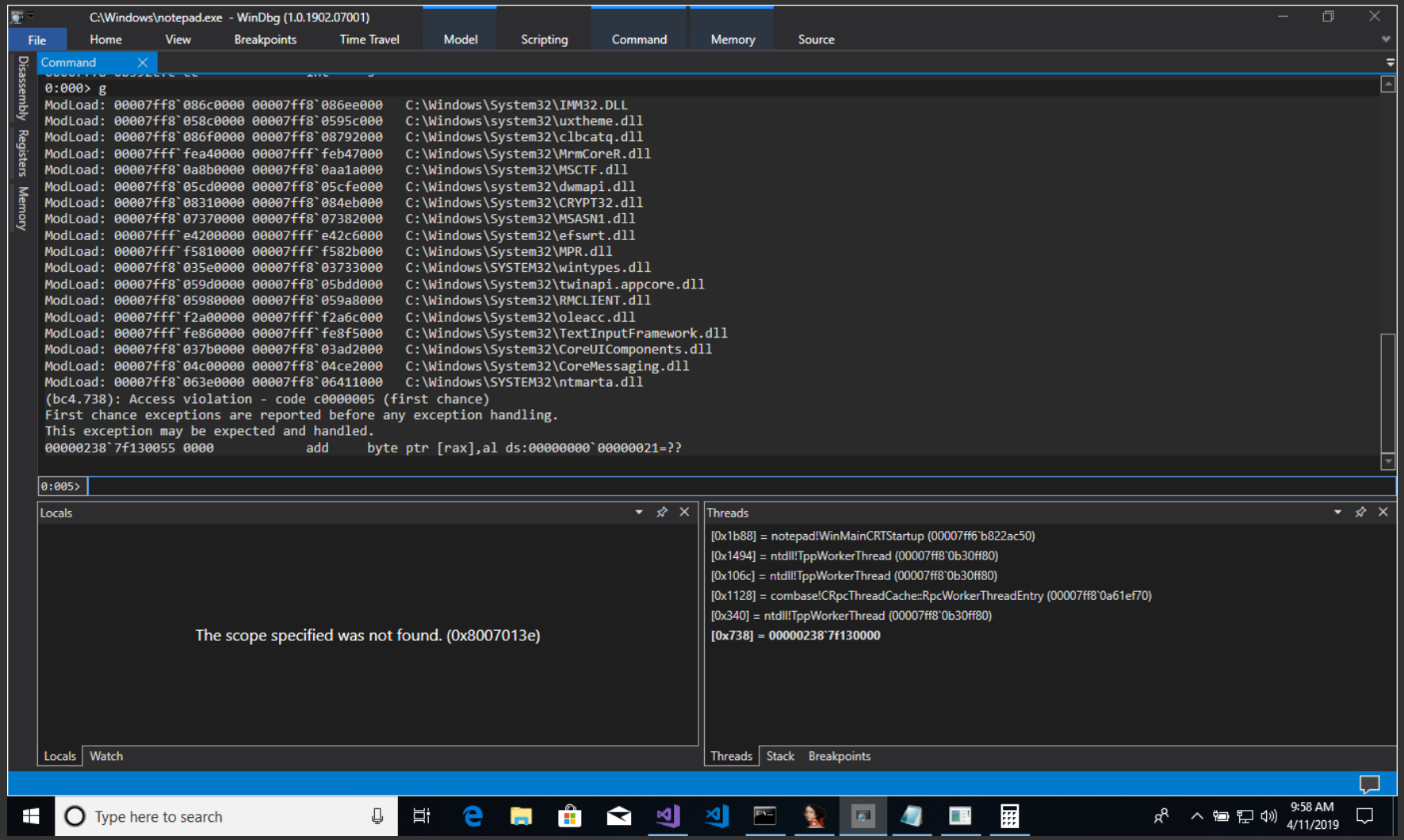
Hit the play button a few times to get through all the breakpoints.

We should now see calculator spawn and windbg should now light up about a few things that happened.



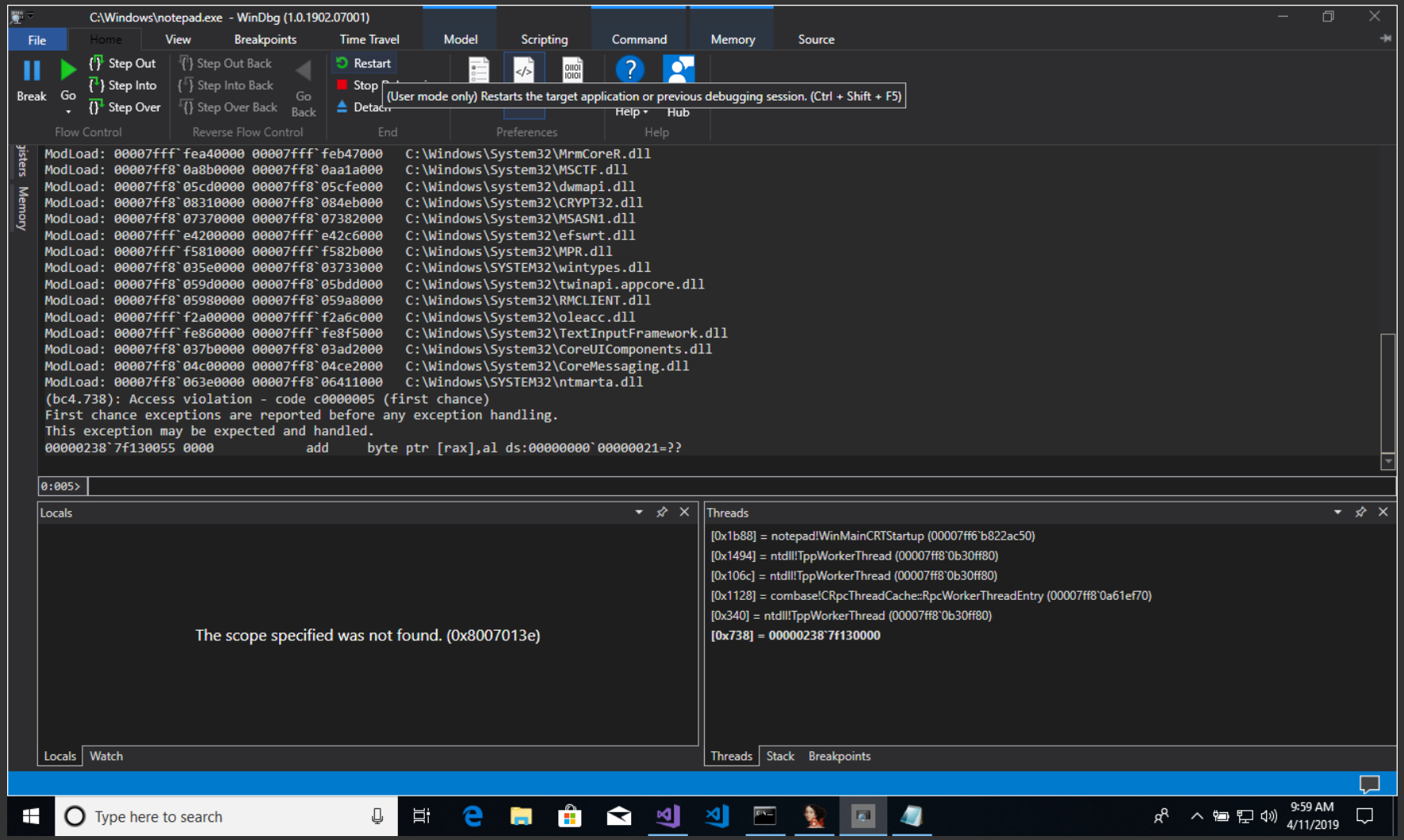
Opening up windbg shows us an Access violation has occurred and the notepad crashed.

This at least tells us why the notepad vanishes shortly before the calculator spawns.



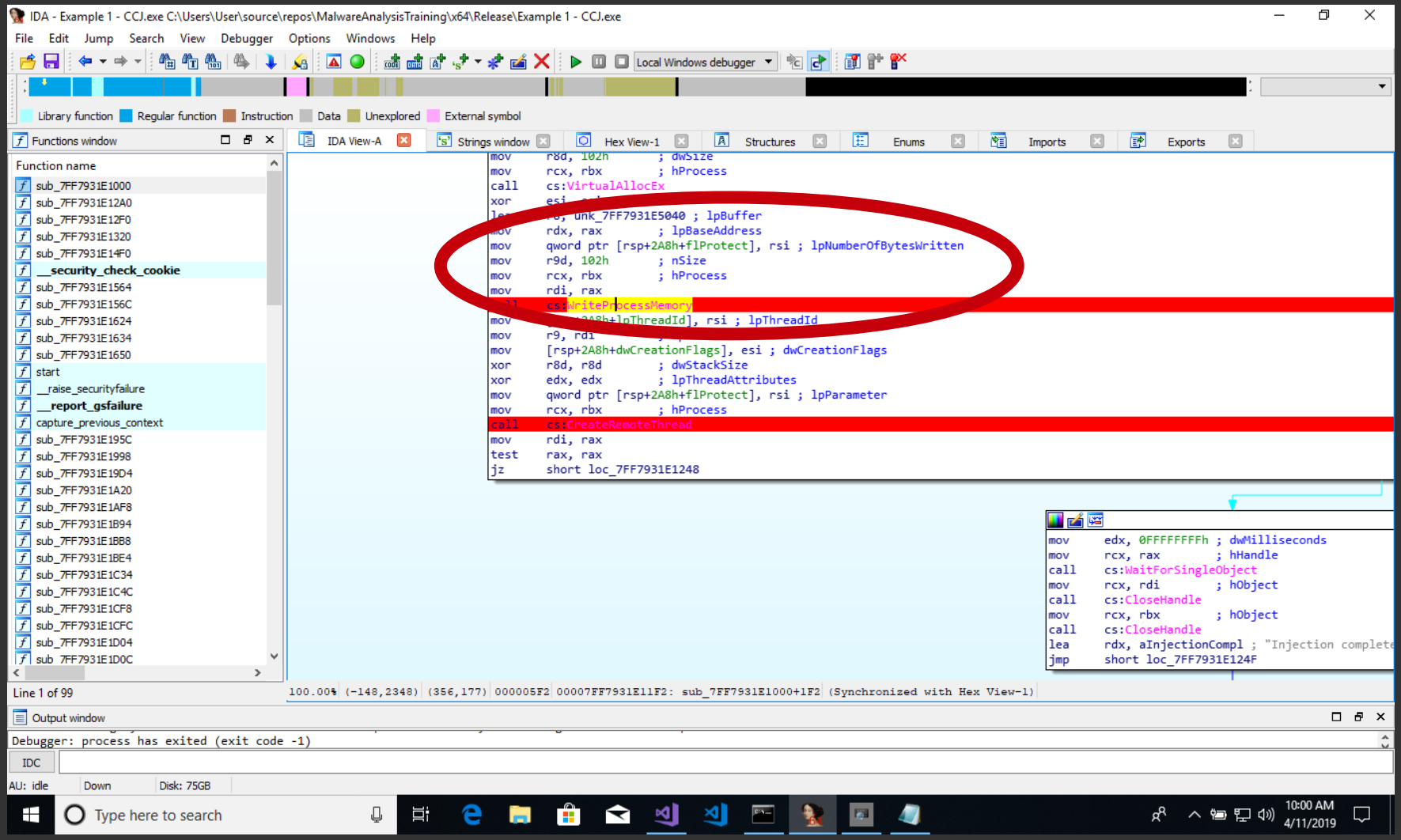
In the middle of the the “Home” view tab there's a restart button for notepad.

Click it to restart notepad, and then hit “Go” to restart the notepad from the start again.



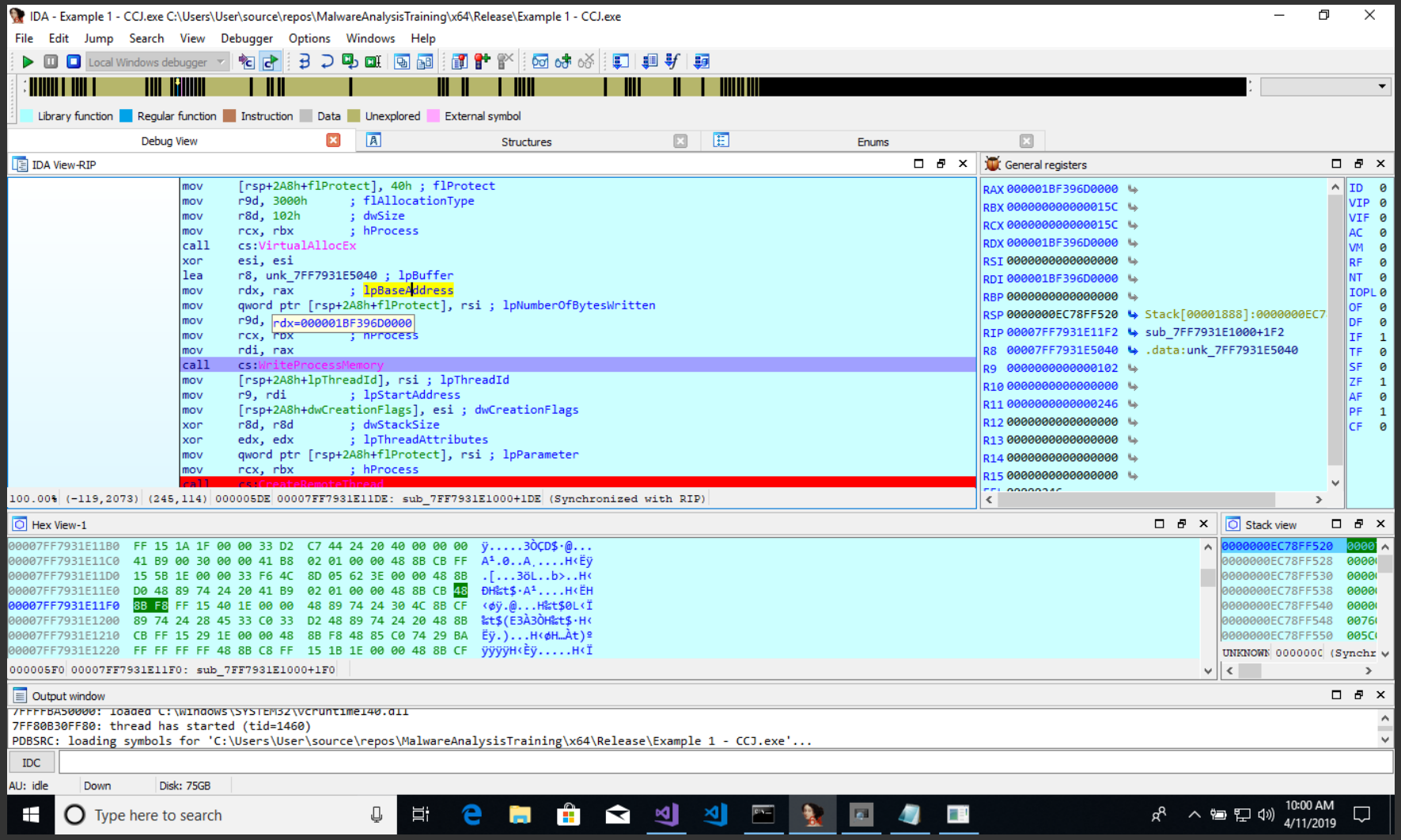
Let's go into IDA and start the malware again, this time however we want to wait at the WriteProcessMemory breakpoint.

The reason, WriteProcessMemory has the location of what is being written into another processes memory. This is held in the “lpBaseAddress” value.



Running the process and continuing past the first breakpoint we end up at WriteProcessMemory.

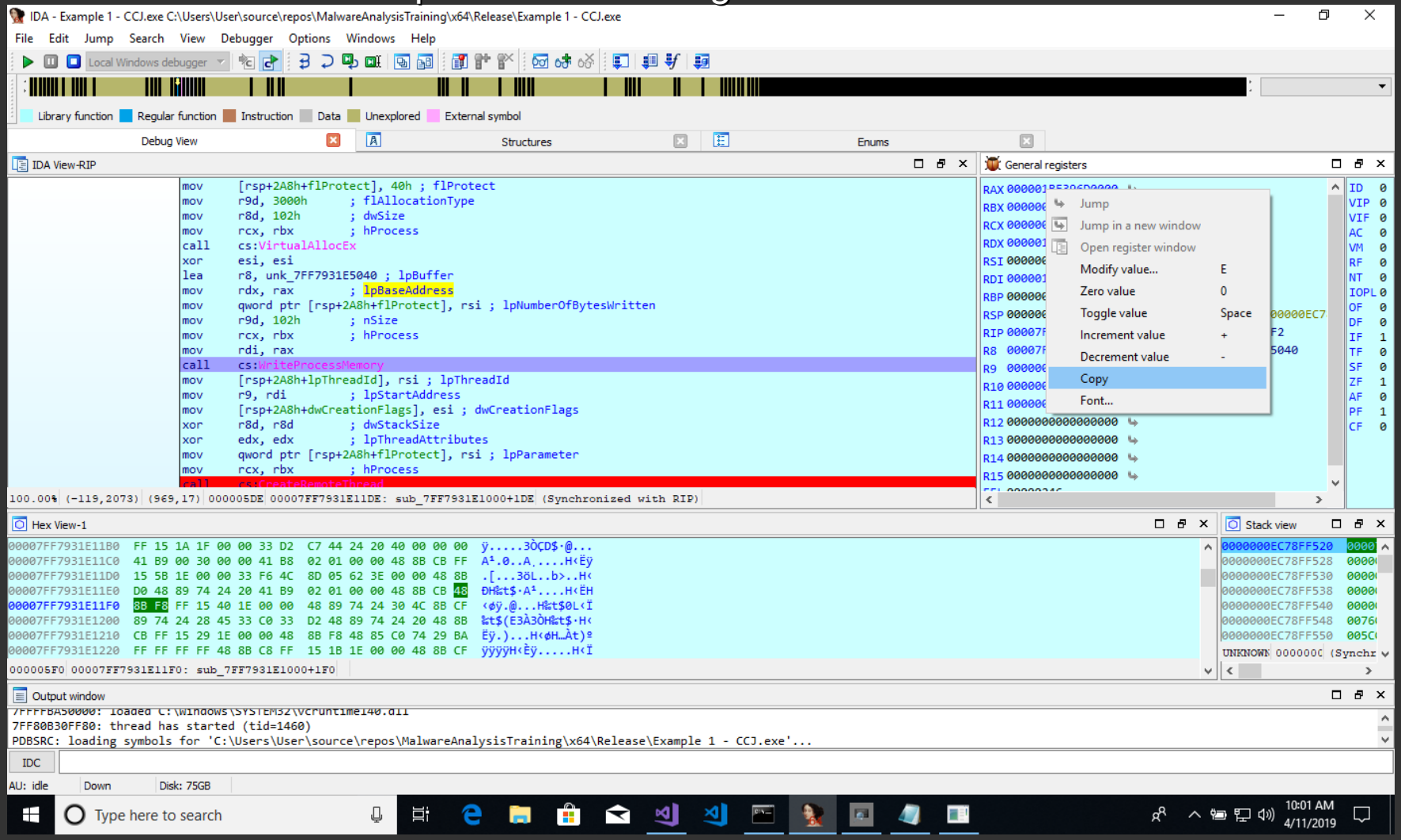
The line with “lpBaseAddress”, highlighted in yellow, shows “mov rdx, rax” which translate to rdx = rax. Ignoring the assembly, it basically means rdx has the “lpBaseAddress”.



The window on the right called “General registers” has the listing of all the registers and what is stored in them currently while debugging.

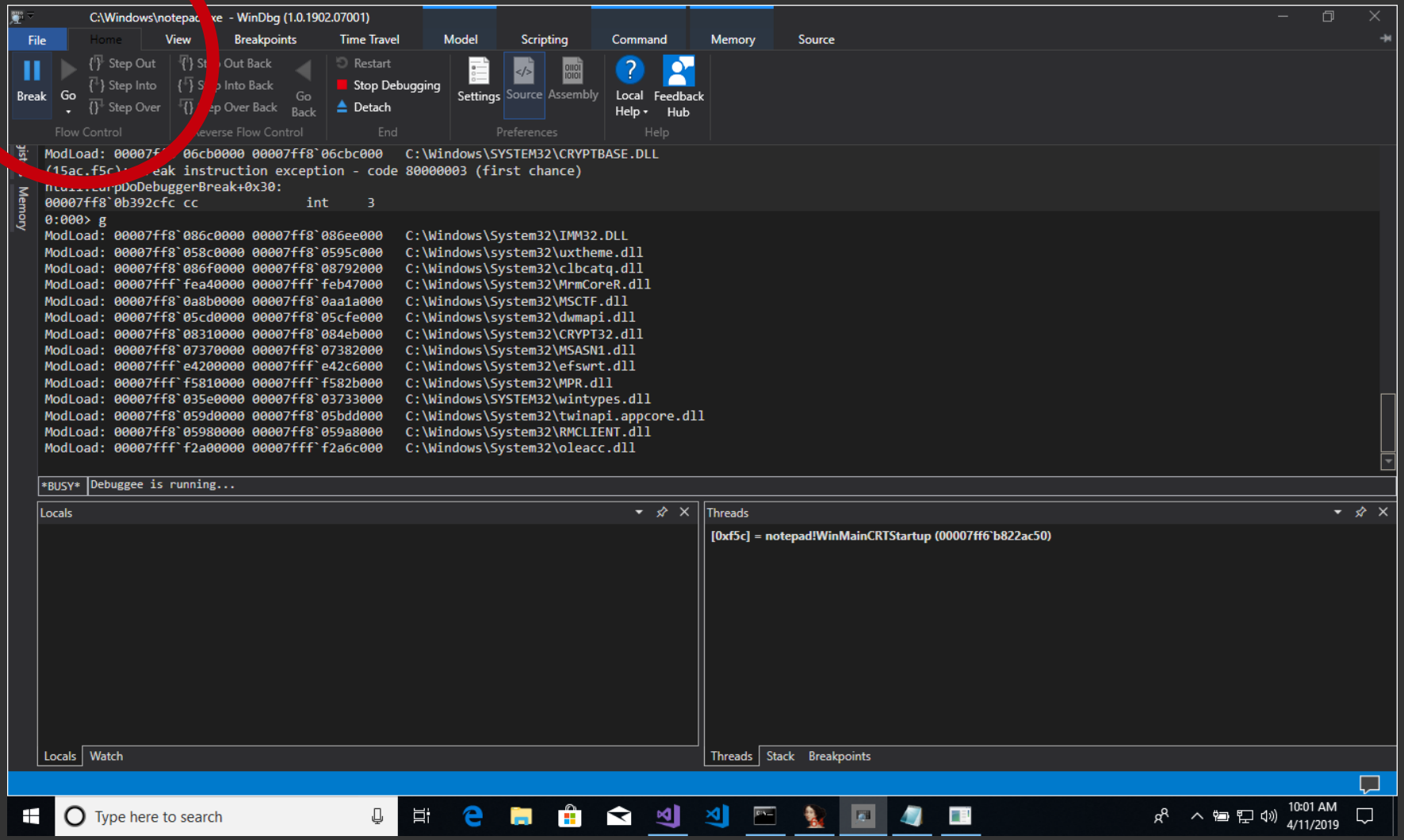
Let's hover over “RDX” and right click and click “Copy”.

We'll need this value to pull over to windbg.



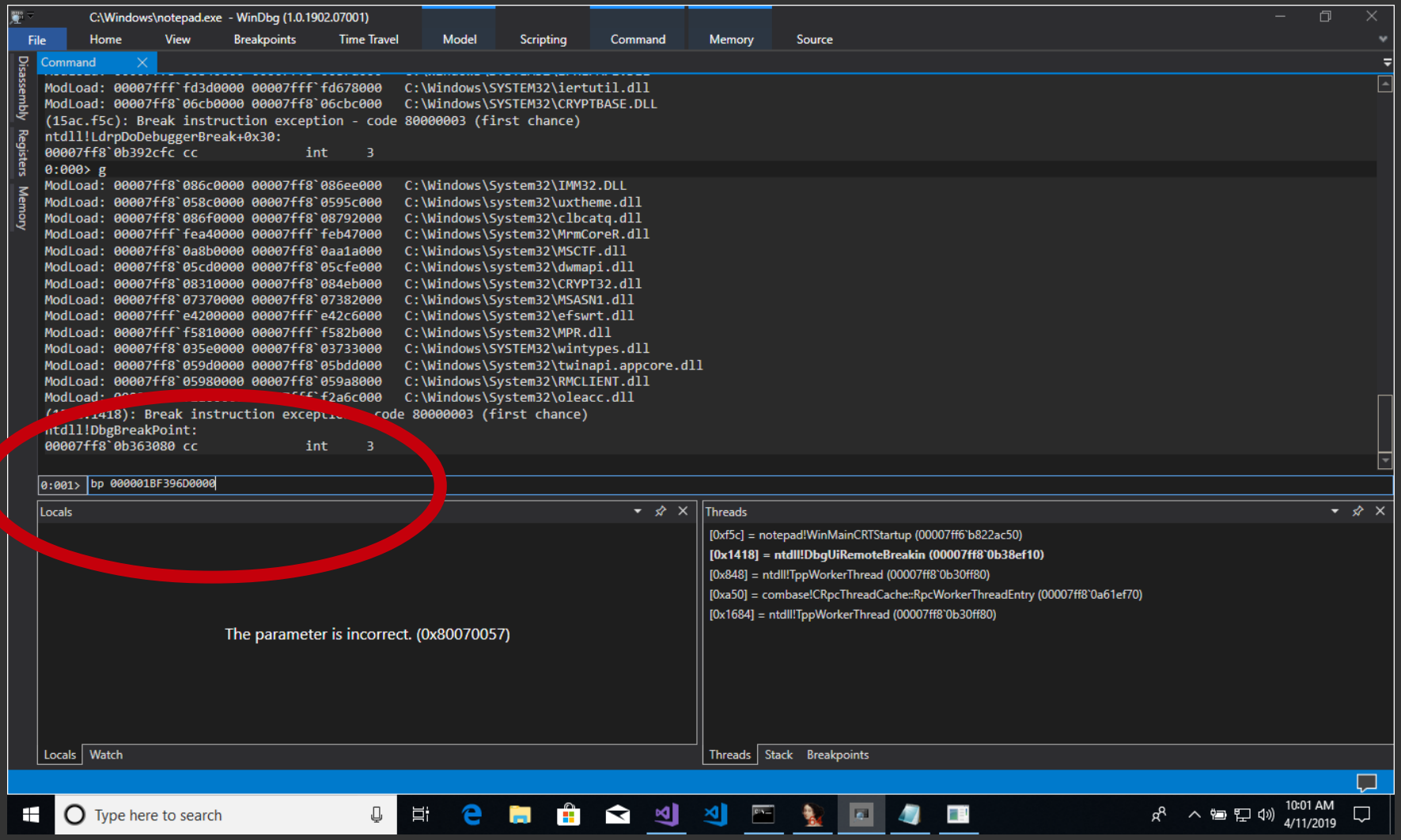
Back in windbg we see a “Break” button next to the “Go” button.

Click the “Break” button to stop the notepad.

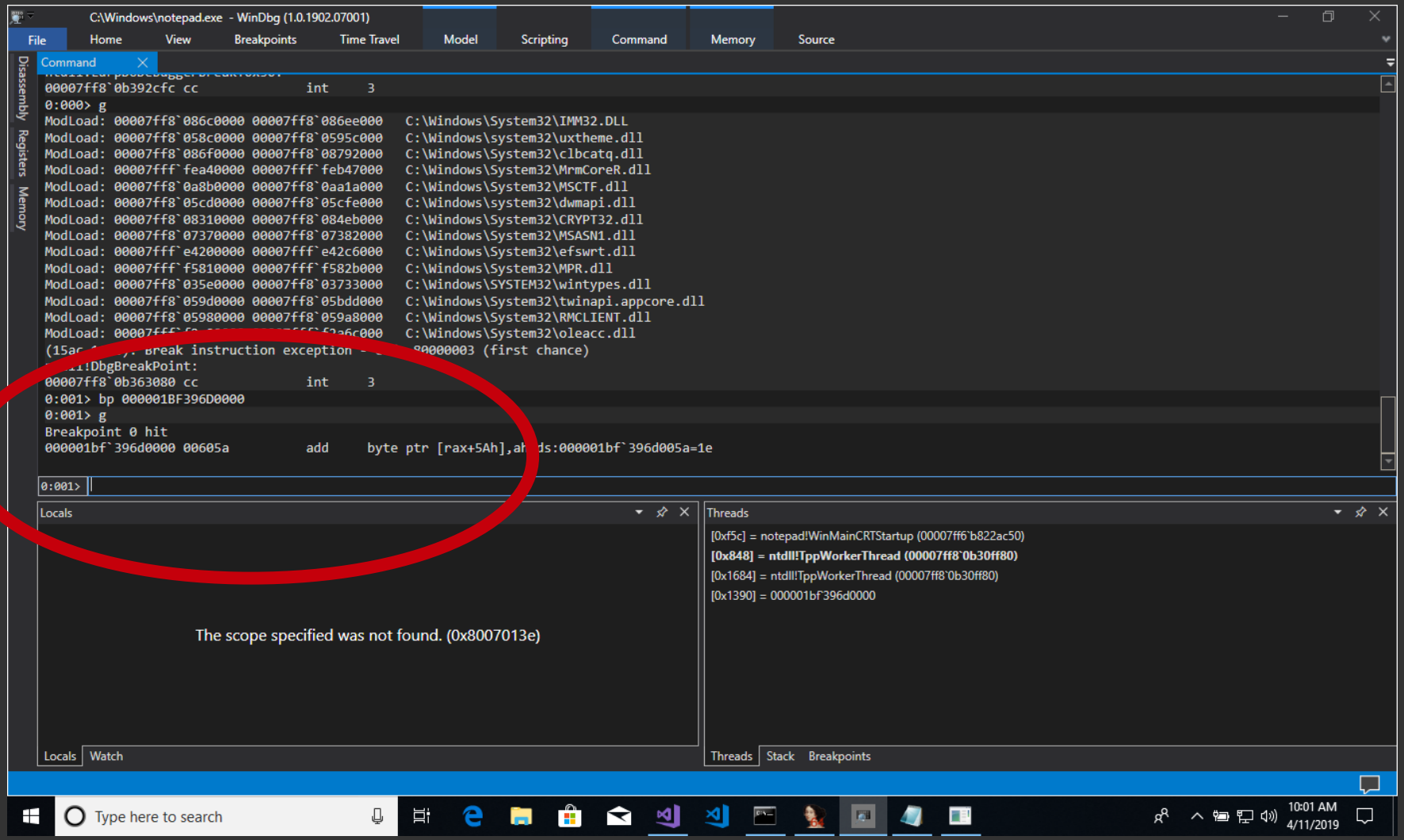


We're going to create a breakpoint in windbg now.

Typed “bp” in the console window and paste the value of “RDX” captured from IDA with a space in between them into the console.

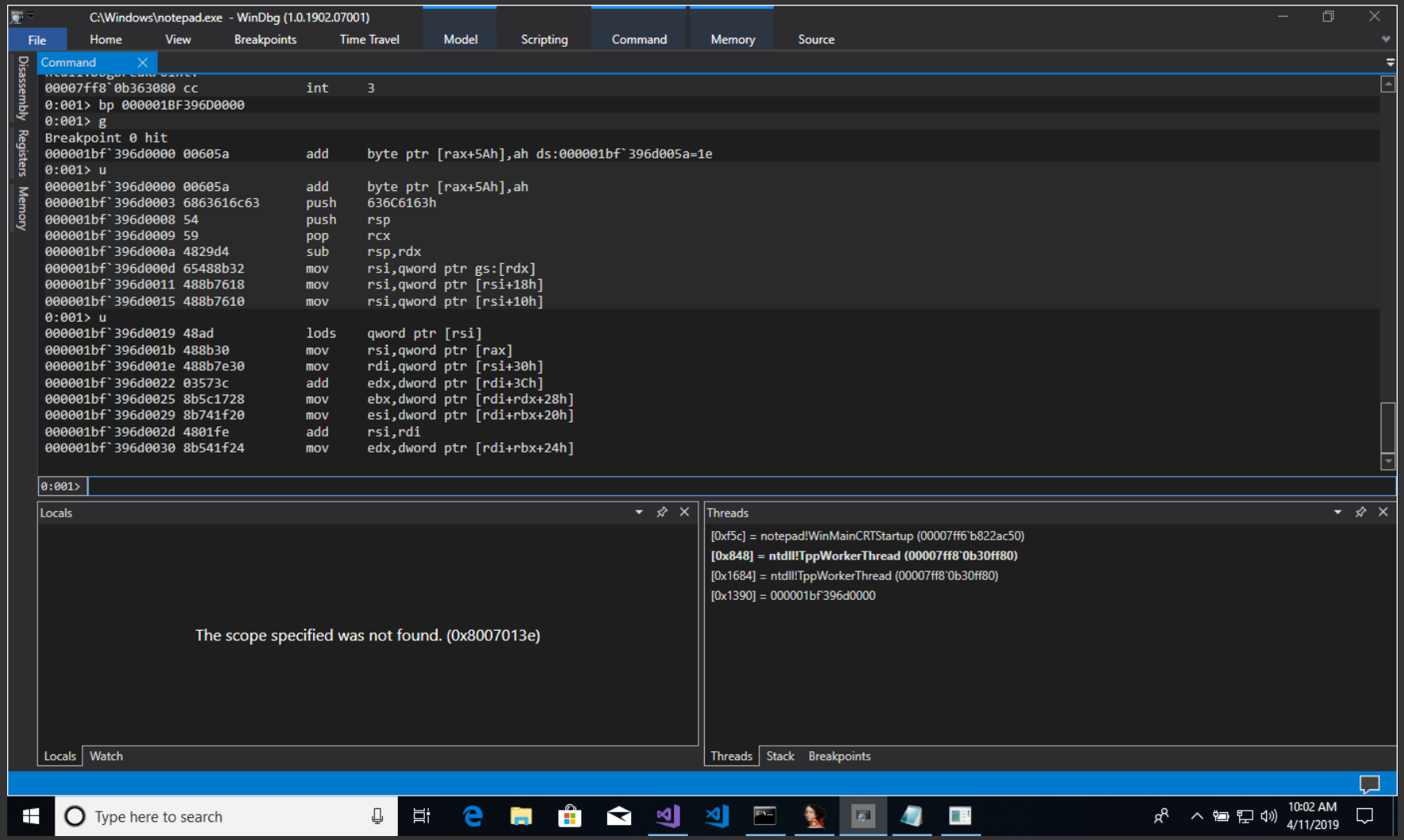


Now hit “Go” again after making the breakpoint. Go back to IDA and continue running the program, and when you come back to windbg again you should see the breakpoint has been hit.



Let's type into the console terminal now in windbg “u”. “u” stands for unassemble in windbg and allows us to see what code is where we are stuck at.

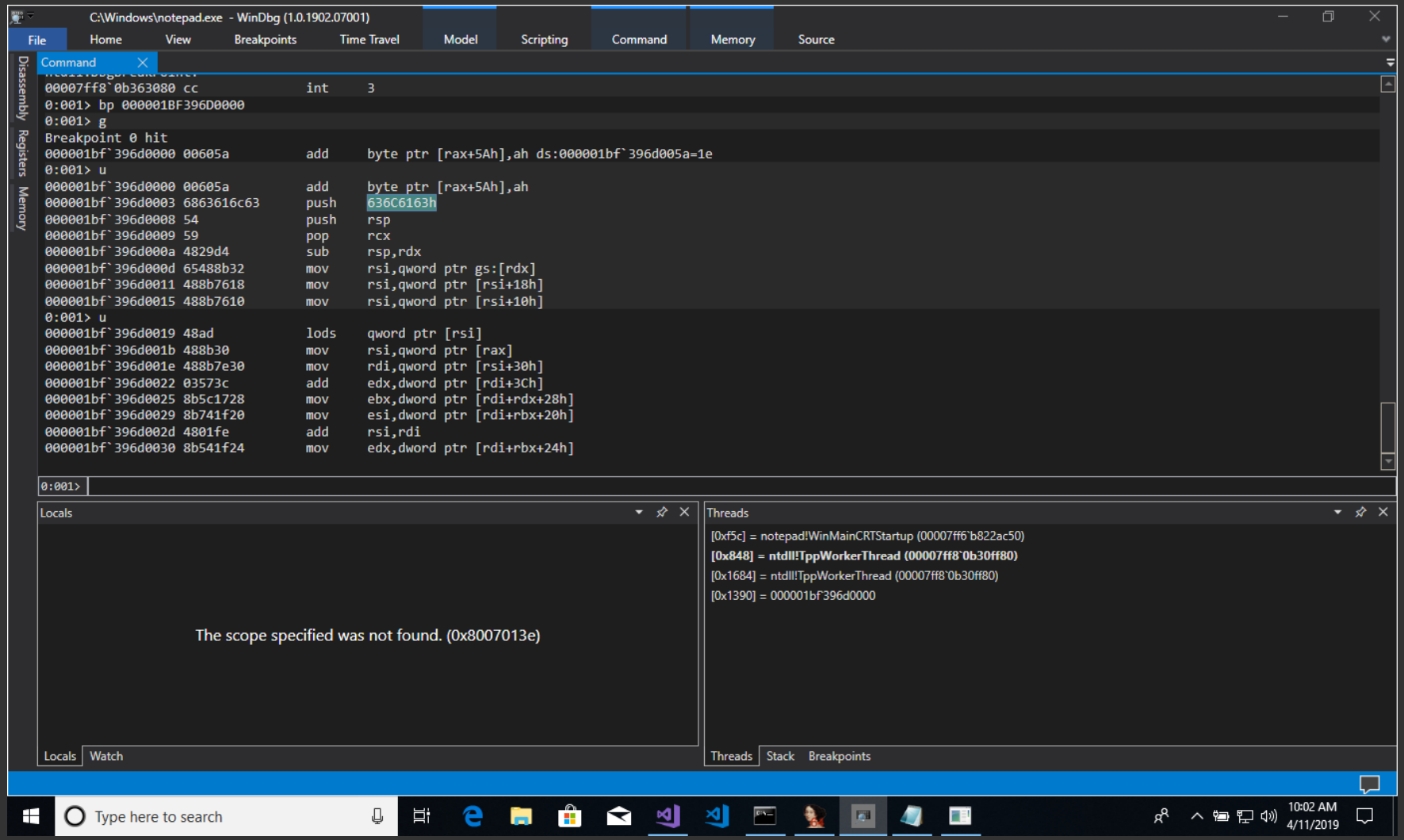
Looks like valid assembly, which means the WriteProcessMemory function is writing code into the notepad.



Now let's take a step back and do some static analysis real fast.

Let's extract anything that looks like strings being held as numbers.

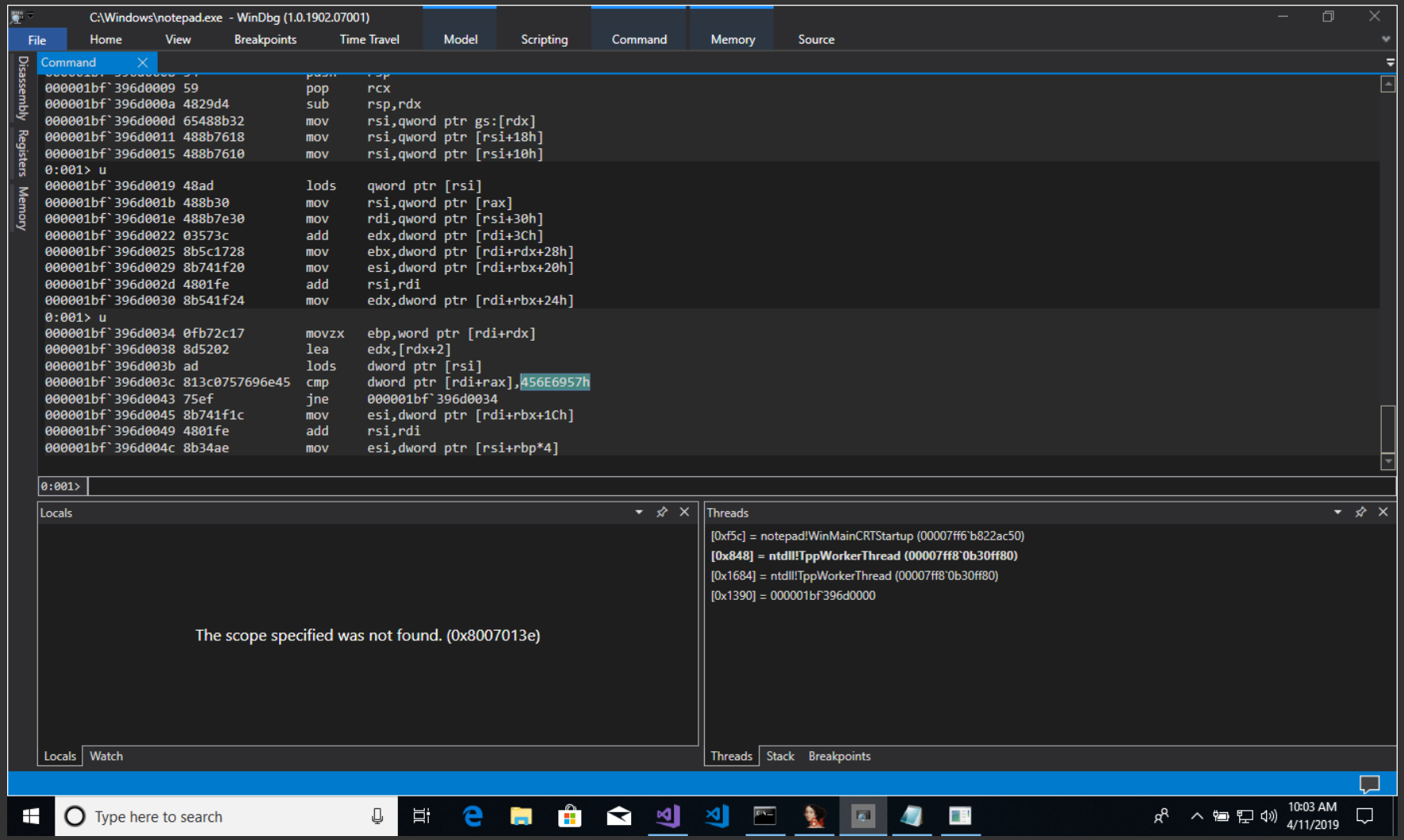
I see 636C6163h



I also see later after typing “u” in again. 456E6957h

Using a hex decoder we see the strings “clac” and “EinW”.

From experience this tells me this is code for a “WinExec(“calc”)” payload.
So now we've generalized the payload and its behavior



Example 1 - End

- At this point we've generalized the malware/binary enough from a start to end perspective.
- Any further analysis and we'd be wasting time on specifics.
- Your goal in every analysis step should be to turn the 100% of surface area to analyze to only 10% in each step.